



Vakgroep Informatietechnologie (INTEC)

Programmeren

Prof. Dr. Ir. Filip De Turck

2^e Bachelor Computerwetenschappen
2^e Bachelor Elektrotechniek
Faculteit Ingenieurswetenschappen en Architectuur
Academisch jaar 2017–2018

©Filip De Turck, alle rechten voorbehouden.

Voorwoord

Deze cursus bouwt verder op de inleidende opleiding Informatica uit de eerste bachelor. In dit opleidingsonderdeel wordt enerzijds de kennis van de onderliggende programmeertalen verruimd door het aanleren van programmeerconcepten en methodologieën, aan de hand van algoritmen, algoritmische strategieën en gegevensstructuren.

Dit opleidingsonderdeel kadert in het geheel van de bachelor-opleiding Ingenieurswetenschappen doordat het een aantal technische grondslagen in de breedte behandelt. Op die manier sluit het aan bij voorgaande opleidingsonderdelen die eerder gericht zijn op programmeervaardigheden, en bereidt het de student(e) voor op volgende opleidingsonderdelen die enerzijds eerder methodologisch georiënteerd zijn en anderzijds op het uitvoeren van realistische ontwikkelingsprojecten in de eindfase van de opleiding.

Volgende onderdelen van deze cursus kunnen onderscheiden worden:

1. Overzicht software ontwikkeling: de verschillende fазen en hun betekenis, de verschillende types van software ontwikkelingsprocessen.
2. Proceduraal programmeren (aan de hand van C): incrementeel aangebracht t.o.v. de voorkennis, met nadruk op functies, macro's, modulariteit, preprocessor, array's en pointers en abstracte datatypes.
3. Hybride talen zoals C++: eveneens incrementeel aangebracht t.o.v. de voorkennis, dus o.a. gebruik van pointers, referentietypes versus de value semantiek, operator overloading, overerving en meervoudige overerving, het onderscheid tussen interface en implementatie, polymorfisme, generiek programmeren en dataabstractie, de STL bibliotheek, exception handling.
4. Platformen en technieken voor software ontwikkeling in grote projecten met verschillende programmeurs: Concurrent Version Systems, generatie en gebruik van Make-bestanden, BDD (Eng.:Behavior Driven Design), TDD (Eng.:Test Driven Development), SPLE (Eng.:Software Product Line Engineering), en technieken voor ontwikkeling van betrouwbare software.
5. Situering van hedendaags belangrijke software technologieën, met enkele voorbeelden (Ruby, Web Services, Middleware, Aspect-orientatie), waarbij de taal Objective-C uitgebreid aan bod komt.

CHAPTER 0. VOORWOORD

6. Een project (in groepen van 2 studenten): waarbij voornamelijk het gebruik van programmeeromgevingen zoals C/C++ grondig aan bod komt.

Als voorkennis voor dit vak wordt het volgende verondersteld: een goede kennis van een eerste programmeertaal (bij voorkeur een object-georiënteerde taal), een initiële kennismaking met de principes van object-oriëntatie, enige kennis van computerarchitectuur (de vereiste voorkennis kan aangebracht worden in dit vak zelf) en ervaring in het gebruik van een computer. Na het volgen (en succesvol beëindigen) van dit vak zal de student(e):

- een goed overzicht hebben van de diverse paradigma's voor software-ontwerp,
- meerdere programmeertalen behoorlijk beheersen,
- in staat zijn om probleemoplossend te denken door inzicht in basis datastructuren en algoritmen,
- inzicht hebben in beschikbare platformen voor programmering van softwareprojecten.

Naast de hoorcolleges, zijn er ook oefeningsessies in de PC-klassen voorzien. Beoordeling van project en practica is equivalent met 1,5 studiepunten (dus 5 punten van de 20 of 25% van de eindevaluatie in 1e examenperiode): practica tellen mee voor 2 punten van de 20 en het project (in groepen van typisch 2 studenten) telt mee voor 3 punten van de 20.

Het examen is een open boek examen, gedeeltelijk op papier (waarbij uitleg en verklaringen gevraagd worden of gevraagd wordt om fouten in een programma op te sporen) en gedeeltelijk aan PC (waarbij gevraagd wordt om een concreet programma te schrijven en er gepeild wordt naar het potentieel aan probleemoplossend vermogen van de student(e), aan hand van de aangereikte technieken en methodes).

We wensen jullie veel succes met dit vak en een leerrijk semester. Voor vragen en suggesties, kunnen jullie steeds volgende e-mail alias gebruiken: *pgm@lists.ugent.be* en kan je altijd terecht op Minerva (forum, document veel gemaakte fouten, dat telkens bijgewerkt wordt, etc.).

Gent, Februari 2018.

Inhoudsopgave

Voorwoord	i
1 Inleiding Software Ontwikkeling	1
1.1 Software ontwikkeling: rollen	1
1.2 Software ontwikkeling: fasen	2
1.3 Software ontwikkeling: tijdsverdeling	3
1.4 Software ontwikkelingsprocessen	3
1.4.1 Plan en documenteer	3
1.4.2 Waterval versus spiraal-model	3
1.4.3 RUP (Eng.:Rational Unified process)	4
1.4.4 Agile Programming	4
1.4.5 Extreme Programming (XP)	4
1.4.6 Test gedreven ontwikkeling	4
1.5 Situering van dit vak	5
I Datastructuren en Algoritmen	7
2 Datastructuren en Algoritmen	9
2.1 Array's	10
2.1.1 Elementaire sortering	10
2.1.2 Geavanceerde sortering	11
2.2 Geschakelde Lijsten - Linked Lists	13
2.2.1 Beperking array's	13
2.2.2 Lineaire lijst - linear list	13
2.2.3 Circulaire lijst - circular list	14
2.2.4 Dubbelgeschakelde lijst - double linked list	14
2.2.5 Multi-lijst - multi list	14
2.3 Boomstructuren - Trees	15
2.3.1 Definitie	15
2.3.2 Doorlopen van boomstructuren	16
2.3.3 Binaire zoekbomen - Binary Search Trees	17
2.3.4 Gebalanceerde zoekbomen - balanced search trees	22

INHOUDSOPGAVE

2.4	Hopen - Heaps	25
2.4.1	Wachtlijnen met prioriteit - priority queues	25
2.4.2	Heap datastructuur	26
2.4.3	Heap algoritmen	26
2.4.4	HeapSort	28
2.5	Grafen - Graphs	29
2.5.1	Definitie	29
2.5.2	Voorstelling van grafen	29
2.5.3	Doorlopen van grafen	30
2.6	Hashtabellen - Hashtables	30
2.6.1	Sleuteltransformaties - hashing	30
2.6.2	Sleuteltransformatiefuncties - hash functions	30
2.6.3	Oplossen van indexconflicten	31
2.6.4	Herschalen van de hashtabel - rehashing	32
2.7	Gecombineerde datastructuren - combined datastructures	33
II	Software Ontwikkeling in C	35
3	Situering Programmeertaal C	37
3.1	Korte historiek	37
3.2	Interpretatie versus compilatie	37
3.3	Uitvoeringssnelheid versus ontwikkelsnelheid en ontwikkelgemak	38
3.4	Situering toepasbaarheid	38
3.5	Programmeeromgeving	39
3.6	Belangrijk	39
4	C Taalelementen	41
4.1	Syntaxregels en taalelementen	41
4.1.1	Backus-Naur Formulering (BNF)	41
4.1.2	Commentaar	42
4.1.3	Programmeerstijl: commentaar	43
4.1.4	Identifiers (namen)	44
4.1.5	Constanten	44
4.2	Operatoren en precedentie	44
4.2.1	Operatoren en punctuatoren	44
4.2.2	Precedentieregels en associativiteit	45
4.3	Variabelen	45
4.3.1	Declaratie van gegevens	46
4.3.2	Uitdrukkingen (Eng.:expressions)	46
4.3.3	Opdrachten (Eng.:statements)	46
4.3.4	Assignatie-opdracht	47
4.3.5	Samentrekking-operatoren	47
4.3.6	Auto-(in/de)crement operatoren	47

INHOUDSOPGAVE

4.4	Controle uitdrukkingen	48
4.5	Fundamentele datatypes	49
4.5.1	Data type modifiers	49
4.5.2	Constante gehele en reële types	50
4.5.3	Typeconversie	50
4.5.4	Cast	52
4.6	Struct's ipv klasses	52
4.6.1	struct declaratie	52
4.6.2	C-idoom	53
4.6.3	Toegang tot gegevens	53
4.6.4	operator =	54
4.6.5	Initialisatie van een struct	54
4.6.6	Struct's: samenvatting	55
4.7	Union's	55
4.8	Elementaire I/O	57
4.8.1	uitvoer: printf functie	57
4.8.2	invoer: scanf functie	57
4.8.3	Macro's getchar en putchar	58
4.9	Wiskundige functies	59
5	C Functies en Macro's	61
5.1	Concept functies in C	61
5.2	Programma modulariteit	62
5.2.1	Concept modules	62
5.2.2	Werking C compiler	63
5.2.3	Linker	63
5.3	Definitie van functies	63
5.4	Declaratie van functies	64
5.4.1	Traditionele C-syntax	66
5.4.2	Uitgebreider voorbeeld	66
5.5	Oproepen van functies: call by value	67
5.6	Storage classes en scope	68
5.6.1	Name hiding/masking	68
5.6.2	Storage classes	69
5.7	Enkelvoudige naamruimte voor functies en globale variabelen	70
5.8	Functienaam overloading	71
5.9	Macro's	71
5.9.1	Concept	71
5.9.2	Definitie	71
5.9.3	Macro's zonder argumenten	72
5.9.4	Gebruik van # bij macro's	72
5.9.5	Gebruik van ## bij macro's	73
5.10	Functies met willekeurig aantal argumenten	73

INHOUDSOPGAVE

6 C Preprocessor	75
6.1 Werking preprocess	75
6.2 Conditionele compilatie	75
6.2.1 Faciliteit tijdens ontwikkeling	77
6.2.2 Portabiliteit	77
6.2.3 Bescherming van headerbestanden	78
6.3 C preprocessor: voorgedefinieerde macro's	79
6.4 <code>assert</code> macro	80
6.5 Compile-time checks: <code>#error</code>	80
7 Rijen en Wijzers	81
7.1 Rijen (Eng.:Arrays) in één dimensie	81
7.1.1 Concept	81
7.1.2 Initialisatie	82
7.1.3 Karakter-rijen	82
7.1.4 C-idioom	83
7.1.5 Arrays als functie-argumenten	83
7.2 Wijzers (Eng.:Pointers)	84
7.2.1 Concept	84
7.2.2 Operatoren & en *	84
7.2.3 Wijzertypes	84
7.2.4 NULL waarde	84
7.2.5 Conversie-karakter	85
7.2.6 Voorbeeld	85
7.2.7 Relationale operatoren	86
7.2.8 Meervoudige wijzers	86
7.2.9 Pointer arithmetica	86
7.2.10 Generieke wijzer-types	87
7.2.11 Verband array's en pointers	87
7.2.12 Oefening	88
7.3 Call by reference	89
7.3.1 Principe	89
7.3.2 Voorbeeld	90
7.3.3 Access type modifiers	91
7.3.4 Array als functie-argument: gebruik <code>const</code>	93
7.4 Dynamisch geheugenbeheer	93
7.4.1 Toewijzen van geheugen - memory allocation	94
7.4.2 Vrijgeven van geheugen - releasing memory	94
7.4.3 Hergebruik van geheugen - reuse of memory	95
7.4.4 Dynamisch geheugenbeheer: fouten en problemen	95
7.4.5 Generieke code dankzij dynamisch geheugenbeheer	96
7.5 Strings	97
7.5.1 Concept	97

INHOUDSOPGAVE

7.5.2	Conversie-karakter	97
7.5.3	String versus <code>char []</code>	97
7.5.4	String functies	98
7.5.5	Geformatteerde I/O van/naar string	98
7.5.6	Voorbeeld: vergelijken van strings	98
7.5.7	Belangrijk - Vermijden van buffer overflows	99
7.6	Multidimensionele rijen	105
7.6.1	Declaratie	105
7.6.2	Initialisatie	105
7.6.3	Adressering	105
7.6.4	Meerdimensionale rijen: wijzerrijen	106
7.6.5	Meerdimensionale rijen: programma-argumenten	106
7.6.6	Meerdimensionale rijen: dynamische rijen	107
7.7	Functie-argumenten	107
7.7.1	Principe	107
7.7.2	Voorbeeld 1: automatisch testen van functies	109
7.7.3	Voorbeeld 2: tabulatie van functiewaarden	109
7.7.4	Functie-argumenten: rijen van functiewijzers	110
7.8	Struct's in combinatie met wijzers	111
7.8.1	Struct's en pointers	111
7.8.2	Struct's: wijzervelden	112
7.8.3	Struct's: soorten kopieën	113
7.8.4	Oefening: diepe kopie	115
7.8.5	Struct's en functies	115
7.8.6	Struct's: geneste structuren	116
8	Andere verschillen met Java	119
8.1	<code>const</code> in plaats van <code>final</code>	119
8.2	<code>inline</code> functies	119
8.3	Afwezigheid van excepties	120
8.4	Type aliasing	120
8.5	<code>main()</code> functie	120
8.6	Standaard bibliotheek	121
8.7	Bitvelden en -operatoren	122
8.8	Opsommingen	123
9	Abstracte Datatypes	125
9.1	Definitie	125
9.2	Implementatie in C	125
9.3	Voorbeeld: stapel (Eng.:stack)	126
9.3.1	Situering	126
9.3.2	Bewerkingen	126
9.3.3	Interface	126
9.3.4	Implementatie	127

INHOUDSOPGAVE

9.3.5 Gebruik	128
9.3.6 Uitbreiding	129
III Software Ontwikkeling in C++	131
10 C++ als uitbreiding van C	133
10.1 C++ historiek	133
10.2 Van abstract datatype naar klasse	133
10.3 Online informatie	135
10.4 Naamruimten	135
10.5 Output en input in C++: I/O streams	136
10.5.1 Output: << operator	136
10.5.2 Input: >> operator	137
10.5.3 Manipulatie van bestanden in C++	138
10.6 C++ types	139
10.6.1 Fundamentele types	139
10.6.2 enum in C++	139
10.6.3 Type casting	140
10.6.4 Referenties	140
10.7 Default functie-argumenten	141
10.8 Dynamisch geheugenbeheer in C++	142
11 Klassen in C++	145
11.1 C++ programma ontwerp	145
11.1.1 Interface specificatie	145
11.1.2 Klasse implementatie	146
11.1.3 Applicatie programma	147
11.1.4 Compilatie en linking	147
11.2 Constructoren en destructor	147
11.3 Initialisatie en allocatie van objecten	150
11.4 Friend functies en klassen	151
11.4.1 Friend functies	151
11.4.2 Friend klassen	152
11.4.3 Friend concept in een object-georiënteerde taal	153
12 Overloading van Functies en Operatoren	155
12.1 Overloading van functies	155
12.2 const methoden	155
12.3 Overloading van operatoren	156
12.3.1 Definitie en voorbeeld	156
12.3.2 Drie opties voor declaratie en implementatie	158
12.3.3 Overzichtstabel	158
12.3.4 Postfix en prefix operatoren	159

INHOUDSOPGAVE

12.3.5 Andere operatoren	159
12.3.6 Code voorbeelden	160
12.4 Operator overloading met friends	163
13 Overerving en Polymorfisme	165
13.1 Basisconcept overerving	165
13.2 Afgeleide klassen	165
13.2.1 Constructoren van afgeleide klassen	166
13.2.2 Het sleutelwoord <code>protected</code>	167
13.2.3 Herdefinitie van member-methoden in afgeleide klassen	167
13.2.4 Copy constructor	168
13.2.5 Destructoren in afgeleide klassen	169
13.2.6 Toekenning-operator =	170
13.2.7 <code>protected</code> en <code>private</code> overerving	171
13.2.8 Meervoudige overerving	172
13.3 Polymorfisme	173
13.3.1 Concept virtuele functies	173
13.3.2 Abstracte basisklassen	176
13.3.3 Toekenning afgeleide klassen	177
13.3.4 Toekenning pointers naar klassen	178
13.3.5 Datastructuren met polymorfe objecten	179
13.3.6 Casting	179
13.3.7 Virtuele destructoren	180
14 Templates	181
14.1 Functie templates	181
14.1.1 Introductie	181
14.1.2 Functie template syntax	181
14.1.3 Meerdere types als parameter	182
14.1.4 Aanbevolen aanpak	183
14.1.5 Toegelaten parameter-types in templates	183
14.2 Klasse templates	183
14.2.1 Overzicht	183
14.2.2 Klasse templates binnen functie templates	184
14.2.3 Aanbevolen aanpak voor ontwerp van klasse templates	185
14.3 Instantiatie en compilatie van templates	185
14.4 Template type definities	186
14.5 Templates en overerving	186
14.6 Oefening	186
15 Standard Template Library (STL)	187
15.1 Inleiding	187
15.2 Voorbeeld: <code>vector</code> container	187
15.2.1 Code voorbeeld	187

INHOUDSOPGAVE

15.2.2 Constructoren	189
15.2.3 Operaties op een <code>vector</code>	189
15.2.4 <code>vector</code> : iteratoren	190
15.2.5 Algoritmen	192
15.3 Functie objecten	192
15.4 Types containers	193
15.5 Sequentiële containers	193
15.5.1 <code>vector</code>	193
15.5.2 <code>deque</code>	194
15.5.3 <code>list</code>	194
15.6 Associatieve containers	194
15.6.1 Inleiding	194
15.6.2 <code>set</code> en <code>multiset</code> containers	195
15.6.3 Hulpklasse template <code>pair</code>	195
15.6.4 <code>map</code> container	196
15.7 Iteratoren	196
15.8 Online referenties	197
15.9 Uitgebreid code voorbeeld	197
15.9.1 Originele versie	198
15.9.2 Verbeterde versie	199
16 Datastructuren in C++	201
16.1 Gelinkte lijsten	201
16.2 Boomstructuren	201
16.3 Hopen	201
16.4 Grafen	201
16.5 Hashtabellen	201
IV Software Ontwikkeling: Platformen en Technologieën	203
17 Concurrent Version Systems (CVSs)	205
17.1 Definitie CVS	205
17.2 Operaties	205
17.2.1 <code>checkout</code> -operatie	205
17.2.2 <code>commit</code> -operatie	206
17.2.3 <code>update</code> -operatie	207
17.2.4 Oplossen van conflicten	207
17.3 Goed gebruik van CVS	208
17.4 Tagging, Branching, Merging	209
17.4.1 Tagging	209
17.4.2 Branching	209
17.4.3 Merging	210
17.5 Bestandsheading - Sleutelwoorden	210

INHOUDSOPGAVE

17.6 CVS implementaties	211
17.7 Referenties	211
18 Make-bestanden	213
18.1 Situering	213
18.2 Compiler opties	214
18.3 Programma <code>make</code> en make-bestanden	215
18.4 Variabelen	216
18.5 Automatische variabelen	216
18.6 Condities	217
18.7 Doelen - Eng.:Targets	218
18.8 Standaardregels	219
18.9 Generische make-bestanden	219
19 Hedendaags Belangrijke Software Technologieën	223
19.1 Java Technologieën	223
19.1.1 Standaard Editie versus Enterprise Editie	223
19.1.2 Prestatie evaluatie	223
19.1.3 JVM tuning	224
19.2 Middleware	224
19.3 Web services	225
19.4 Ruby	225
19.5 Aspect-oriëntatie	225
19.6 Software voor mobiele toestellen	226
20 Ontwikkeling van Betrouwbare Software	227
20.1 BDD (Eng.:Behavior Driven Design)	227
20.2 SPLE (Eng.:Software Product Line Engineering)	227
20.3 Software as a Service - SaaS	228
20.4 Ontwikkelingsproces	228
20.4.1 Scrum	228
20.4.2 Tools voor code validatie	228
20.4.3 Geautomatiseerd testen	228
20.4.4 Issue and project tracking software	229
20.4.5 Documentatie-beheer	229
20.4.6 Continue Integratie	229
20.4.7 Release workflows	229
20.5 Eigenschappen van goed software ontwerp	229
20.5.1 Data aggregation	229
20.5.2 Zero configuration	229
20.5.3 Daemon monitoring and startup	230
20.5.4 Dashboard applicatie	230
20.5.5 Product feedback	230

INHOUDSOPGAVE

21 Software ontwikkeling in de praktijk	231
21.1 Overzicht Software-ontwikkelingsproces	231
21.1.1 Opdracht bestuderen	232
21.1.2 Vereisten identificeren	232
21.1.3 Opdeling in componenten	233
21.1.4 Uitwerking architectuur	234
21.1.5 Coderen	235
21.1.6 Evaluatie	236
21.2 Implementatie Aspecten	237
21.2.1 Van design tot code	237
21.2.2 Goede coding practices	237
21.2.3 Gebruik van IDE	239
21.2.4 Collaboratie	240
V Software Ontwikkeling in Objective-C	245
22 Objective-C	247
22.1 Korte historiek	247
22.2 Boodschappen - Eng.:Messages	247
22.3 Interfaces en Implementaties	249
22.3.1 Interface	249
22.3.2 Implementatie	251
22.3.3 Aanmaken van objecten	254
22.3.4 Vernietigen van objecten	254
22.4 Compileren en Linken	255
22.5 Protocollen - Eng.:Protocols	256
22.6 Doorsturen - Eng.:Forwarding	257
22.7 Categorieën - Eng.:Categories	257
22.8 Overerving in Objective-C	258
22.8.1 Initialisatie van objecten van afgeleide klassen	259
22.8.2 Verwijdering van objecten van afgeleide klassen	259
22.8.3 <code>isa</code>	259
22.9 Verschillen met C++	259
22.10 Geheugenbeheer in Objective-C	260
22.10.1 Verschillende opties	260
22.10.2 Bezit van een object - Eng.:Object Ownership	261
22.10.3 Aanmaken van objecten via klasse methoden en via instantie methoden	262
22.10.4 Boodschappen voor geheugenbeheer	262
22.10.5 Geheugenbeheer: vuistregels	263
22.10.6 Autorelease pool	263
22.10.7 Geheugenbeheer bij gebruik van containers	264
22.10.8 Geheugenbeheer bij accessor methoden	264

INHOUDSOPGAVE

22.10.9 Gebruik van properties	264
VI Appendices	267
A Dynamic Memory Management in C	269
A.1 Allocation	269
A.2 Memory structure	270
A.3 Important rules	271
A.4 Illustrative scenarios	272
B Overzicht Eigenschappen C++11	277
B.1 Inleiding	277
B.2 Automatische type-afleiding en decltype	277
B.3 Uniforme initialisatiesyntax	278
B.4 Deleted en Defaulted functies	279
B.5 nullptr	280
B.6 Delegerende constructoren	280
B.7 Rvalue Referenties	280
B.8 Nieuwe Smart Pointer klassen	283
B.9 Anonieme functies: Lambda expressies	284
B.10 C++11 Standard Library	285
B.10.1 Nieuwe algoritmes	286
B.11 Threading bibliotheek	286
B.11.1 Het lanceren van threads	287
B.11.2 Beschermen van data	288
B.11.3 Ondersteuning voor thread-events	289
B.12 Conclusie	290
C C Traps and Pitfalls	291
D C++ Tips and Traps	322

INHOUDSOPGAVE

Hoofdstuk 1

Inleiding Software Ontwikkeling

1.1 Software ontwikkeling: rollen

In een complex software project is het noodzakelijk dat de betrokken personen een specifieke rol op zich nemen. Vooral in bedrijven onderscheidt men de rollen van werknemers. Een typische rolverdeling wordt weergegeven in tabel 1.1, tesamen met de Engelse vertaling van de rollen.

De functionele analist zorgt voor een analyse van de opdracht en onderhandelt met de klanten i.v.m. de vereiste functionaliteit van de applicatie. De klant geeft aan hoe hij de applicatie wenst te gebruiken en welke elementen in de applicatie dienen opgenomen te worden. De vereisten kunnen functioneel zijn (Eng.: functional requirements, of kortweg *functionals* genoemd): vastleggen van de vereiste functionaliteit (bijv. te voorziene knoppen, menu's, berekeningen, visualisaties) van de applicatie. De overblijvende vereisten worden niet-functionele vereisten (Eng.: non-functional requirements, of kortweg *non functionals*) genoemd. Voorbeelden van niet-functionele vereisten zijn prestatiekenmerken van de applicatie (bijv. hoe snel de applicatie een resultaat geeft, hoeveel klanten de applicatie gelijktijdig kunnen gebruiken) of ingebouwde veiligheidsmaatregelen (Eng.: security measures).

Rol (Ned.)	Role (Eng.)
Functionele Analist	Functional Analyst
Software Architect	Software Architect
Ontwikkelaar	Developer
Software Tester	Software Tester
Projectleider	Project Leader
Kwaliteitsbeheerder	Quality Manager
Integratiebeheerder	Integration Manager
Verkoper	Sales Person

Tabel 1.1: Verschillende rollen bij het software ontwikkelingsproces

In sommige firma's zijn er ook bedrijfsanalisten (Eng.: business analysts) actief, deze zorgen voor analyse van de bedrijfsprocessen (Eng.: business processes) en identificeren in welk bedrijfsproces welke software kan ingepast worden in overleg met de klant.

De architect zorgt voor het conceptuele ontwerp van de applicatie en de implementatiekeuzes (welke software technologieën en bibliotheken gebruikt worden).

De ontwikkelaars zorgen voor de eigenlijke codering van de applicatie, in overleg met de software architect. Volgende specialiteiten van ontwikkelaars kunnen bijvoorbeeld onderscheiden worden: graphic designer, user interface designer, web designer, web developer, database administrator, support technician.

De software tester is verantwoordelijk voor het grondig testen van de ontwikkelde onderdelen en het geven van feedback en schrijven van documenten. Belangrijk is dat de software tester niet bij de codering van de onderdelen betrokken is, zodat hij onafhankelijk en onbeïnvloed zijn oordeel kan geven.

De projectleider zorgt voor toezicht op de timing en de gebruikte financiële middelen en de communicatie met de klanten tijdens de ontwikkeling, installatie en ter plaatse evaluatie van de software.

De kwaliteitsbeheerder zorgt bij het testen voor nuttige feedback in verband met de bruikbaarheid van de applicatie en verzorgt ook kwaliteitscontrole bij het ontwikkelproces (bijv. voorzien van duidelijke commentaar in de code, duidelijke documentatie).

De integratiebeheerder zorgt voor de coördinatie van de integratie (i.e. het samenvoegen tot een geheel) van de ontwikkelde onderdelen (modules, componenten, bibliotheken, eventueel ook hardware) en het grondig testen hiervan.

De verkoper zorgt voor goede contacten met de klanten en is enkel in de initiële fase bij een software ontwikkelingsproces betrokken.

Fase (Ned.)	Phase (Eng.)	betrokken personen
Vereisten	Requirements	Functionele analist, Verkoper, Architect
Specificatie	Specification	Architect, Functionele analist
Ontwerp	Design	Architect
Codering van Modules	Module coding	Architect, Ontwikkelaar
Testen van Modules	Module testing	Software Tester, Ontwikkelaar
Integratie	Integration	Integratiebeheerder, Ontwikkelaar
Onderhoud	Maintenance	alle rollen

Tabel 1.2: Indicatie kosten verschillende ontwerpsfasen

1.2 Software ontwikkeling: fasen

Voor de ontwikkeling van software worden 7 fasen onderscheiden, zoals in tabel 1.2 weergegeven. Deze fasen worden niet noodzakelijk sequentieel doorlopen. In de rechterkolom van de tabel wordt ook weergegeven welke personen in elke fase betrokken zijn.

Fase (Ned.)	percentage kost
Vereisten	2%
Specificatie	5%
Ontwerp	6%
Codering van Modules	5%
Testen van Modules	7%
Integratie	8%
Onderhoud	67%

Tabel 1.3: Indicatie kosten verschillende ontwerpsfasen

De typische kosten gespendeerd door firma's in elk van de fasen, wordt weergegeven in tabel 1.3. Het valt op dat onderhoud een flink deel van het budget kan innemen.

1.3 Software ontwikkeling: tijdsverdeling

De kosten in tabel 1.3 geven een idee van de hoeveelheid werk, maar voor de volledigheid wordt in tabel 1.4 ook de typische tijdsverdeling tussen de fasen weergegeven (zonder de onderhoudsfase omdat deze sterk projectafhankelijk is).

Vereisten en specificatie	18%
Ontwerp	19%
Implementatie (Codering + Testen)	34%
Integratie	29 %

Tabel 1.4: Indicatie tijdsverdeling verschillende ontwerpsfasen

1.4 Software ontwikkelingsprocessen

Onderstaande ontwikkelingsprocessen worden zeer vaak gebruikt in de praktijk.

1.4.1 Plan en documenteer

Dit is de klassieke manier, waarbij eerst een gedetailleerd ontwerp gemaakt wordt en dan pas aan de implementatie begonnen wordt. Elke iteratie wordt nauwkeurig gepland en gedocumenteerd.

1.4.2 Waterval versus spiraal-model

Dit zijn twee varianten van het "Plan en documenteer-proces. In het waterval-model worden de 7 bovenstaande fasen lineair doorlopen, terwijl in het spiraal-model meer

iteratief gewerkt wordt. Volgende fazen worden in het spiraal-model onderscheiden:

1. vastleggen van de doelstellingen en randvoorwaarden van de iteratie,
2. evaluatie van alternatieven, en vastleggen en zoveel mogelijk oplossen van risico's,
3. ontwikkelen en verifiëren van het prototype van deze iteratie,
4. de volgende iteratie plannen.

1.4.3 RUP (Eng.:Rational Unified process)

Dit proces combineert elementen van het waterval en het spiraal-modellen: de verschillende fases tijdens de iteraties worden volgens een waterval-model doorlopen. Er wordt ook meer rekening gehouden met de business aspecten dan enkel de technische aspecten.

1.4.4 Agile Programming

Hierbij worden lichtgewicht cycli voorgesteld in het software ontwikkelingsproces met veel interacties tussen de ontwikkelaars en met de klanten. Er wordt meer nadruk gelegd op het teamwerk van het ontwikkelingsteam en een frequente en vlotte samenwerking met de klanten. In plaats van telkens een voorgedefinieerd plan te volgen, dient er snel te kunnen aangepast worden als blijkt dat dit nodig is.

1.4.5 Extreme Programming (XP)

Is een belangrijk voorbeeld van Agile Programming. Er wordt veel nadruk gelegd op het programmeren per twee, zoveel mogelijk uitvoeren van code reviews, en testen van elkaars code. Verder wordt er slechts gestart met het programmeren van modules wanneer ze echt nodig zijn, en wordt er zoveel mogelijk geanticipeerd op aanpassingen in de vereisten van de klanten.

1.4.6 Test gedreven ontwikkeling

TDD (Eng.:Test-driven development) is gebaseerd op volgend principe: er wordt gestart met het schrijven van test code om een module, die nog moet ontwikkeld of aangepast worden, te testen, vervolgens wordt de minimale code geschreven om de module te realiseren of aan te passen. Vanzodra de geschreven code aan de test voldoet, wordt de code aangepast om aan de kwaliteitstandaarden te voldoen. Het voordeel van deze aanpak is dat de nadruk gelegd wordt op een eenvoudig ontwerp en dat er telkens met vertrouwen kan verdergewerkt worden (aangezien de geschreven code de testen mooi doorstaan heeft).

1.5 Situering van dit vak

De kennis die in dit vak opgebouwd wordt, situeert zich vooral in de ontwerpsfase en implementatiefase (nadruk op codering en testen van de ontwikkelde software). Het komt met andere woorden de opleiding van software architect en software ontwikkelaar ten goede. Bemerk dat de andere rollen meestal geen specifieke opleiding dienen te hebben: men komt meestal in een rol terecht door ervaring opgedaan in vorige projecten, en zich te specialiseren in een bepaald onderdeel tijdens een software ontwikkelingsproces. In deze cursus komen de programmeertalen C en C++ uitgebreid aan bod, en worden ook platformen en hedendaagse software technologieën behandeld. Vermits de student een basiskennis van de programmeertaal Java heeft, worden de verschilpunten met Java aangegeven. Het bijhorende handboek *De programmeertaal C* (Eng.: *A Book on C*) is een internationaal erkend referentiewerk, waar tijdens deze cursus regelmatig zal naar verwezen worden.

Hoofdstuk 1: Inleiding Software Ontwikkeling

Deel I

Datastructuren en Algoritmen

Hoofdstuk 2

Datastructuren en Algoritmen

In dit hoofdstuk komen datastructuren en algoritmen aan bod, die zeer veel gebruikt worden in het kader van programmeren. De principes worden uitgelegd: concrete source code met de implementatie en het gebruik van deze datastructuren en algoritmen komt in hoofdstukken van deze cursus aan bod. Belangrijk is om te begrijpen wanneer welke datastructuur aangewezen is en wat de prestatie is van enerzijds de algoritmen en anderzijds de operaties op de datastructuren.

Zes datastructuren en bijhorende algoritmen komen aan bod:

1. Array's
2. Gelinkte lijsten
3. Boomstructuren
4. Hopen
5. Grafen
6. Hashtabellen

De keuze van een datastructuur voor opslag van gegevens geeft weer hoe de gegevens georganiseerd worden in het geheugen (aan de hand van structuren, tabellen, etc.). Deze keuze heeft invloed op de manier waarop operaties op de gegevens kunnen toegepast worden en hoe efficiënt (op gebied van vereiste berekeningen en vereiste hoeveelheid geheugen) deze kunnen uitgevoerd worden. Voorbeelden van operaties zijn: sorteren, zoeken, bijvoegen en verwijderen van elementen, samenvoegen van datastructuren, etc. Belangrijk om op te merken is dat de vermelde datastructuren elementen van **hetzelfde type** bevatten. Verder in de cursus zal aan bod komen welke technieken kunnen aangewend worden om:

- elementen van verschillende types in dezelfde datastructuur te kunnen opslaan (aan de hand van zogenaamde generieke types),

- een datastructuur zodanig te implementeren dat ze voor verschillende types kan gebruikt worden, zonder dat de broncode van de implementatie van de datastructuur zelf dient aangepast te worden (aan de hand van zogenaamde sjablonen (Eng.: templates)).

2.1 Array's

Een array slaat elementen sequentieel op in het geheugen. Belangrijk is dat alle elementen van **hetzelfde type** zijn (zoals hierboven vermeld voor alle datastructuren, die in dit hoofdstuk aan bod komen) en bovendien fysisch in het geheugen allen net naast elkaar opgeslagen worden (geen ruimte ertussen).

2.1.1 Elementaire sortering

Een aantal eenvoudige sorteermethoden worden hier kort herhaald.

Sorteren door selectie - selection sort

Deze methode bestaat er in om in de array eerst het kleinste element te zoeken (selecteren) en vervolgens te verwisselen met het eerste element. Vervolgens wordt in resterende sub-array opnieuw het kleinste element gezocht (geselecteerd) en verwisseld met het eerste element van de sub-array. Dit wordt herhaald tot wanneer de sub-array slechts één element bevat.

Sorteren door invoeging - insertion sort

Hierbij wordt de array gesorteerd door iteratief de elementen van links naar rechts te overlopen en telkens het nieuwe element gesorteerd in de reeds overlopen sub-array te plaatsen. Dit vereist het opruimen van de elementen na het in te voegen element (om plaats te maken voor de invoeging).

Sorteren door borrelen - bubble sort

Deze methode bestaat erin om de array te overlopen van links naar rechts, en elk element te vergelijken met zijn rechterbuur: als deze rechterbuur groter is, worden de elementen omgewisseld. Dit doorlopen en omwisselen met de rechterbuur wordt herhaald tot wanneer geen enkele omwisseling meer plaatsvindt.

Een variant is sorteren door schudden (Eng.: shake sort), waarbij de array van links naar rechts doorlopen wordt, vervolgens van rechts naar links, etc. en telkens het principe van eventuele omwisseling met de buur toegepast wordt.

Prestatievergelijking

Veronderstellen we een array met N elementen. Sortering door selectie vereist

$$1 + 2 + \dots + (N - 2) + (N - 1) \approx \frac{N^2}{2}$$

vergelijkingen en maximaal N omwisselingen. Sorteren door invoeging vereist gemiddeld

$$\frac{1 + 2 + \dots + (N - 2) + (N - 1)}{2} \approx \frac{N^2}{4}$$

vergelijkingen en omwisselingen. Sorteren door borrelen vereist

$$1 + 2 + \dots + (N - 2) + (N - 1) \approx \frac{N^2}{2}$$

vergelijkingen en maximaal evenveel omwisselingen.

2.1.2 Geavanceerde sortering

Twee technieken komen hier aan bod: Quicksort en Mergesort. Beide zijn voorbeelden van verdeel en heers algoritmen.

Quicksort

Bij uitvoering van dit algoritme wordt eerst een partitie-element (ook wel pivot-element genaamd) geselecteerd. Dit partitie-element is bijvoorbeeld het laatste of het middelste element van de array.

Vervolgens wordt dit partitie-element op zijn finale plaats in de array geplaatst. Dit gebeurt door het element gesorteerd in te voegen: links van het partitie-element staan geen elementen met een grotere waarde dan het partitie-element en rechts van het partitie-element staan geen elementen met een kleinere waarde dan het partitie-element.

Vervolgens wordt iedere partitie (deel-array) op dezelfde manier **recursief** gesorteerd.

Mergesort

Het grote verschil met Quicksort is dat geen partitie-element of pivot-element gebruikt wordt om dit reeds op zijn finale plaats te plaatsen en op die manier de array in twee delen te splitsen.

Bij Mergesort wordt eerst de array in twee delen verdeeld door het middelste element te selecteren. Bij een even aantal elementen wordt de array dus in twee gelijke delen gesplitst. Bij een oneven aantal elementen, bevat het ene deel één element meer dan het andere deel.

Vervolgens worden de beide helften **recursief** gesorteerd. De beide helften worden dan nadien in volgorde samengevoegd (Eng.: merged).

Prestatievergelijking

In het slechtste geval is het partitie-element bij Quicksort steeds het kleinste of het grootste element van de array, waardoor er N recursieve oproepen vereist zijn. Het vereiste aantal vergelijkingen in dit geval bedraagt:

$$(N - 1) + (N - 2) + \dots + 1 = \frac{N \times (N - 1)}{2} \approx \frac{N^2}{2}.$$

De prestatie van Quicksort is dus afhankelijk van de volgorde waarin de elementen in de array aanwezig zijn, bijvoorbeeld (i) of de array reeds bijna gesorteerd is, (ii) omgekeerd gesorteerd is, (iii) of er veel gelijke waarden aanwezig zijn.

Er kunnen enkele optimalisaties voor Quicksort bedacht worden:

- Quicksort heeft te veel overhead voor kleine array's, vanaf een zekere ondergrens M voor de grootte van de array kan een elementaire sorteermethode, bijv. sorteren door invoegen, gebruikt worden.
- Quicksort werkt het best als de array telkens in ongeveer twee gelijke delen gesplitst wordt (m.a.w. als het partitie-element telkens goed gekozen wordt). Een veelgebruikte techniek is om het partitie-element te bepalen als de mediaan van drie elementen (in dit geval is de kans groter dat het partitie-element er inderdaad voor zorgt dat de array in twee zo gelijk mogelijke delen gesplitst wordt).

Indien de array niet gesorteerd is (elementen in willekeurige volgorde toegevoegd) en er gepartitioneerd wordt op een willekeurig element, kan men aantonen dat het aantal vergelijkingen ongeveer

$$2 \times N \times \log_2(N)$$

bedraagt en in het meest ideale geval:

$$1.39 \times N \times \log_2(N)$$

bedraagt. Het bewijs van deze uitdrukking ligt buiten het bereik van deze cursus (intuïtief kan men aanvoelen waarom er een $\log_2(N)$ verband is en dit beter presteert dan een N^2 verband).

Bij Mergesort is het samenvoegen (Eng.: merging) en kopiëren van de elementen de flessthals (Eng.: bottleneck). Twee deel-array's van grootte $\frac{N}{2}$ samenvoegen vereist N vergelijkingen. Mergesort heeft

$$N \times \log_2(N)$$

vergelijkingen nodig om een bestand met N elementen te sorteren, maar heeft extra geheugen nodig dat proportioneel is met N . Men gebruikt als regel dat voor kleine array's Mergesort 10% a 15% sneller kan presteren dan Quicksort.

2.2 Geschakelde Lijsten - Linked Lists

2.2.1 Beperking array's

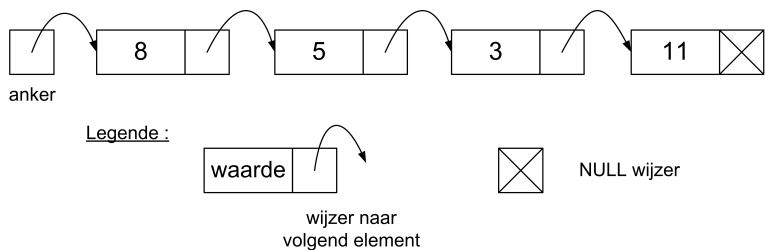
Het gebruik van array's heeft volgende beperkingen:

1. toevoegen/verwijderen van een element vereist kopiëren van alle volgende elementen (behalve indien het laatste element verwijderd wordt of een element achteraan toegevoegd wordt)
2. elementen worden sequentieel opgeslagen: indien er onvoldoende ruimte is voor een uitbreiding, dient de gehele array gekopieerd te worden naar een plaats in het geheugen waar er wel voldoende plaats is voor een uitbreiding.

Omwille van deze redenen wordt vaak geopteerd voor het gebruik van geschakelde lijsten.

2.2.2 Lineaire lijst - linear list

Bij een lineaire lijst, wordt aan elk element een extra variabele geassocieerd, namelijk een wijzer (Eng.: pointer) naar het volgende element in de lijst. Op deze manier wordt de lijst aan elkaar geschakeld (vandaar de benaming). Een wijzervariabele bevat een fysisch adres, in geval van een lineair geschakelde lijst dus het adres van het volgende element. Indien een element meerdere geheugenplaatsen inneemt, bevat de wijzervariabele het adres van de eerste geheugenplaats. Wanneer er geen volgend element is (bijv. op het einde van de lijst of bij een lege lijst), is de conventie dat de wijzervariabele als waarde 0 bevat (alle bits van de wijzervariabele op waarde 0). Een waarde 0 wordt dikwijls ook als NULL genoteerd. Een belangrijke variabele is het **anker** (Eng.: anchor): deze is een wijzervariabele die het adres van het eerste element in de lijst bevat. Figuur 2.1 toont een voorbeeld van een lineaire lijst, die gehele getallen als element bevat.



Figuur 2.1: Voorbeeld van een lineair geschakelde lijst die gehele getallen bevat.

Toevoegen van een element

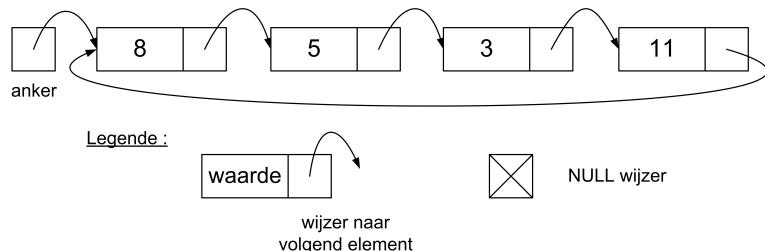
Het toevoegen van een element gebeurt door de wijzervariabele van het vorige element in de lijst te laten wijzen naar het nieuwe element en de wijzervariabele van het nieuwe element te laten wijzen naar het volgende element in de lijst. De volgorde waarin deze beide operaties uitgevoerd worden is belangrijk.

Verwijderen van een element

Het verwijderen van een element gebeurt door de wijzervariabele van het vorige element te laten wijzen naar het volgende element. Op deze manier verdwijnt het element automatisch uit de lijst. Het geheugen dat ingenomen wordt door het element en zijn geassocieerde wijzervariabele, is dan niet meer nodig en kan vrijgegeven worden (zodat het vrijgekomen geheugen tijdens de uitvoering van het programma kan hergebruikt worden).

2.2.3 Circulaire lijst - circular list

Een circulaire lijst is een speciaal geval van een lineaire lijst: de geassocieerde wijzervariabele van het laatste element verwijst opnieuw naar het eerste element. Op deze manier kan de lijst gemakkelijk meerdere malen na elkaar doorlopen worden. Het toevoegen en verwijderen van elementen gebeurt uiteraard op dezelfde manier als bij een lineaire lijst. Figuur 2.2 toont een voorbeeld van een circulaire lijst.



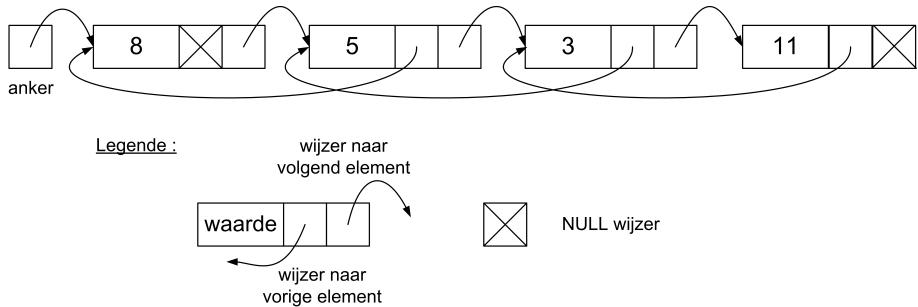
Figuur 2.2: Voorbeeld van een circulaire lijst die gehele getallen bevat.

2.2.4 Dubbelgeschakelde lijst - double linked list

Bij een dubbelgeschakelde lijst worden er met elk element twee wijzervariabelen geassocieerd: een wijzervariabele die het adres van het volgende element bevat (cfr. een lineaire lijst) en een wijzervariabele die het adres van het vorige element bevat. Op deze manier kan men gemakkelijk de lijst in beide richtingen doorlopen. Bovendien kan men eenvoudiger een nieuw element *voor* een bepaald element toevoegen (in vergelijking met een lineaire lijst, waar men een nieuw element *achter* een bepaald element toevoegt) en kan een element verwijderd worden zonder dat eerst het vorige element uit de lijst dient bepaald te worden (zoals in het geval van een lineaire lijst). Figuur 2.3 toont een voorbeeld van een dubbelgeschakelde lijst.

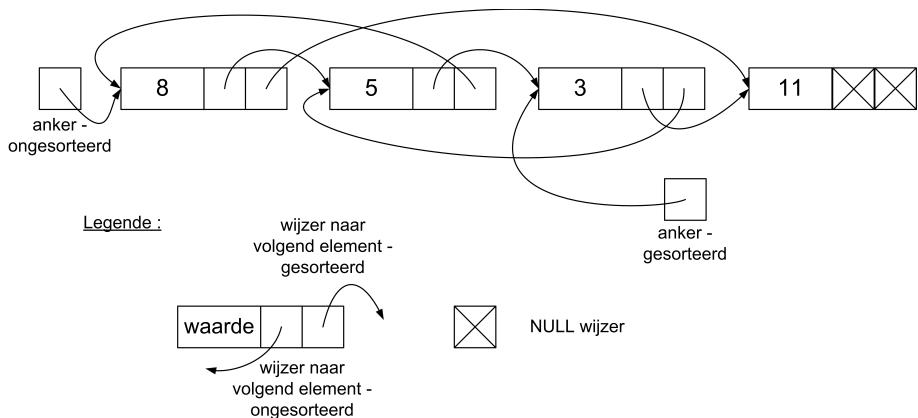
2.2.5 Multi-lijst - multi list

Het principe van een multi-lijst is dat er steeds twee of meerdere wijzervariabelen met elk element geassocieerd worden. Deze wijzervariabelen houden dan meerdere onafhankelijke lijsten bij. Bijvoorbeeld één lijst is de originele input-lijst en de andere lijst is de gesorteerde lijst (waarbij de elementen gesorteerd zijn). Op deze manier kan men op



Figuur 2.3: Voorbeeld van een dubbelgeschakelde lijst die gehele getallen bevat.

een geheugen-efficiënte manier zowel de ongesorteerde als de gesorteerde lijst bijhouden. Figuur 2.4 toont een voorbeeld van een multi-lijst.



Figuur 2.4: Voorbeeld van een multi-lijst die gehele getallen bevat, zowel gesorteerd als ongesorteerd.

2.3 Boomstructuren - Trees

2.3.1 Definitie

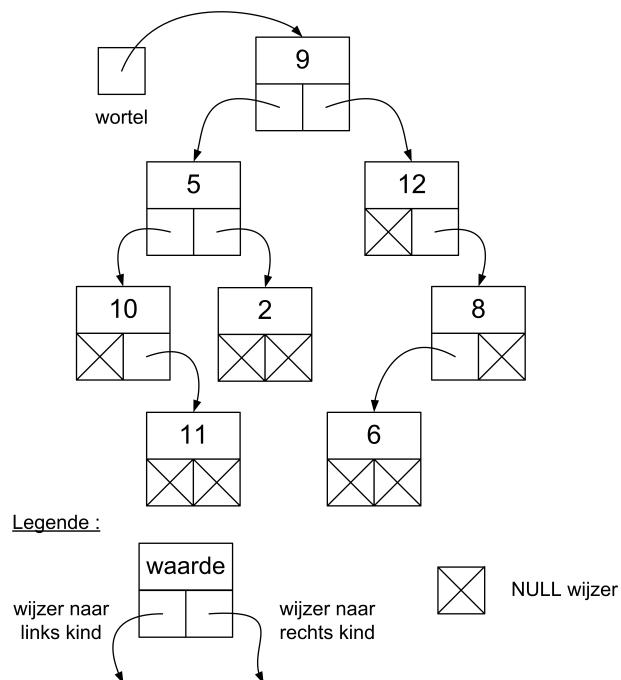
Een boom is een niet lege verzameling van knooppunten (Eng.: vertices, enkelvoud: vertex) en randlijnen (Eng.: edges). Een knooppunt bevat de informatie-elementen en de geassocieerde wijzervariabelen naar de volgende elementen in de boom. Een randlijn is de verbinding tussen twee knooppunten (gerealiseerd door de wijzervariabelen, cfr. bij geschakelde lijsten).

Een pad is gedefinieerd als een verzameling knooppunten die verbonden zijn door een reeks opéénvolgende randlijnen.

Een belangrijke eigenschap van een boomstructuur is dat er precies één pad is tussen twee willekeurige knooppunten. Dit in tegenstelling tot een graaf, waarbij er meerdere

paden kunnen zijn tussen sommige knooppuntenparen en/of er knooppuntenparen bestaan waartussen er geen pad bestaat. Grafen komen aan bod in sectie 2.5.

Een M-waardige boom (Eng.: M-ary tree) is een boom waarbij iedere knoop maximaal M sub-bomen heeft (in een gekende volgorde). Een sub-boom in een M-waardige boom kan uiteraard ook leeg zijn. Een **binaire boom** is een andere benaming voor een 2-waardige boom. Een eigenschap van een binaire boom is dat van elk knooppunt de linker- en rechter sub-bomen ofwel leeg zijn ofwel ook binaire bomen zijn (recursieve eigenschap). Figuur 2.5 toont een voorbeeld van een binaire boom, waarin gehele getallen in een willekeurige volgorde opgeslagen zijn. Net zoals bij geschakelde lijsten, is er een wijzervariabele die het adres van het eerste element bevat. Deze wordt de wortel (Eng.: root) van de boom genoemd. De bovenste node van een sub-boom wordt de ouderknoop (Eng.: parent node) genoemd en de linker- en rechter sub-bomen het linkerkind (Eng.: left child) en het rechterkind (Eng.: right child).



Figuur 2.5: Voorbeeld van een boom die gehele getallen bevat. Belangrijk: dit is **geen** binaire zoekboom, deze komen in sectie 2.3.3 aan bod.

2.3.2 Doorlopen van boomstructuren

Er zijn 4 manieren om een boom te doorlopen.

PreOrder

Recursieve beschrijving: eerst wordt de ouderknoop doorlopen, daarna wordt de linker sub-boom doorlopen (in PreOrder volgorde) en tenslotte wordt de rechter sub-boom doorlopen (eveneens in PreOrder volgorde).

InOrder

Recursieve beschrijving: eerst wordt de linker sub-boom doorlopen (in InOrder volgorde), daarna wordt de ouderknoop doorlopen en tenslotte wordt de rechter sub-boom doorlopen (eveneens in InOrder volgorde).

PostOrder

Recursieve beschrijving: eerst wordt de linker sub-boom doorlopen (in PostOrder volgorde), daarna wordt de rechter sub-boom doorlopen (eveneens in PostOrder volgorde) en tenslotte wordt de ouderknoop doorlopen.

Bovenstaande methoden worden meestal ook recursief geïmplementeerd, maar kunnen eveneens iteratief geïmplementeerd worden (bijvoorbeeld door gebruik te maken van een stapel (Eng.: stack) of wachtrij (Eng.: queue) datastructuur).

LevelOrder

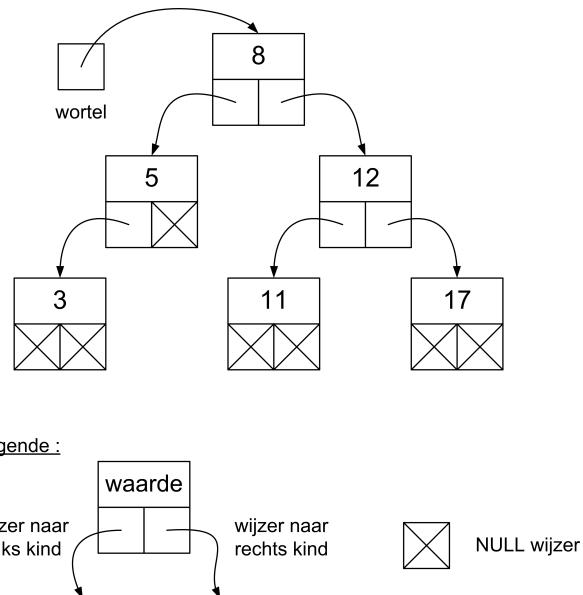
Hierbij wordt eerst de ouderknoop doorlopen, dan van links naar rechts de beide kind-knopen, en vervolgens de knopen op het volgende niveau (alle kleinkind-knopen van de ouderknoop), etc.

2.3.3 Binaire zoekbomen - Binary Search Trees

Een binaire zoekboom wordt gebruikt om elementen **gesorteerd** op te slaan volgens het volgende principe:

- de waarde van de sleutel van iedere knoop is groter dan de waarde van alle sleutels in de linker subboom,
- de waarde van de sleutel van iedere knoop is kleiner dan (of gelijk aan) de waarde van alle sleutels in de rechter subboom.

Figuur 2.6 illustreert de binaire zoekboom datastructuur, waarbij gehele getallen gesorteerd opgeslagen worden, volgens de bovenstaande regels. Een veelgebruikte afkorting voor een binaire zoekboom is BST (Eng.: Binary Search Tree).



Figuur 2.6: Voorbeeld van een binaire zoekboom, die gehele getallen bevat.

Zoeken in een binaire zoekboom - search

Vermits de elementen gesorteerd zijn opgeslagen, kan volgend algoritme toegepast worden om een element met een bepaalde waarde op te zoeken in de binaire zoekboom: vergelijk de waarde van het element met de waarde opgeslagen in de ouderknoop, indien deze kleiner is: ga naar de linker sub-boom en herhaal hetzelfde principe daar, indien de waarden gelijk zijn: element gevonden (wordt *search hit* genoemd), indien de waarde groter is: ga naar de rechter sub-boom en herhaal hetzelfde principe daar. Wanneer een gekozen sub-boom leeg is, stopt het algoritme: element niet gevonden (wordt *search miss* genoemd).

Er wordt dikwijls een recursieve implementatie van dit algoritme gekozen.

Toevoegen van elementen aan een binaire zoekboom - insert

Toevoegen van een element aan een binaire zoekboom is vergelijkbaar met zoeken (sectie 2.3.3). Het element wordt gezocht in de zoekboom, bij een *search miss* wordt het onderaan de boom toegevoegd.

Als er reeds een element met die waarde bestaat, zijn er twee opties: (i) een foutbericht genereren bij een *search hit* (geen duplicate waarden - no duplicate values), (ii) het zoekalgoritme laten lopen tot wanneer een *search miss* tegenkomen wordt (elke *search hit* negeren).

Toevoegen in de wortel - root insertion

Zoals in sectie 2.3.3 beschreven, gebeuren alle toevoegingen in een binaire boom aan de toppen van de boom (i.e onderaan). Voor sommige toepassingen is het aangewezen dat het nieuwe element zich in de wortel bevindt, vermits dit de toegangstijd verkort voor de meest recente knopen.

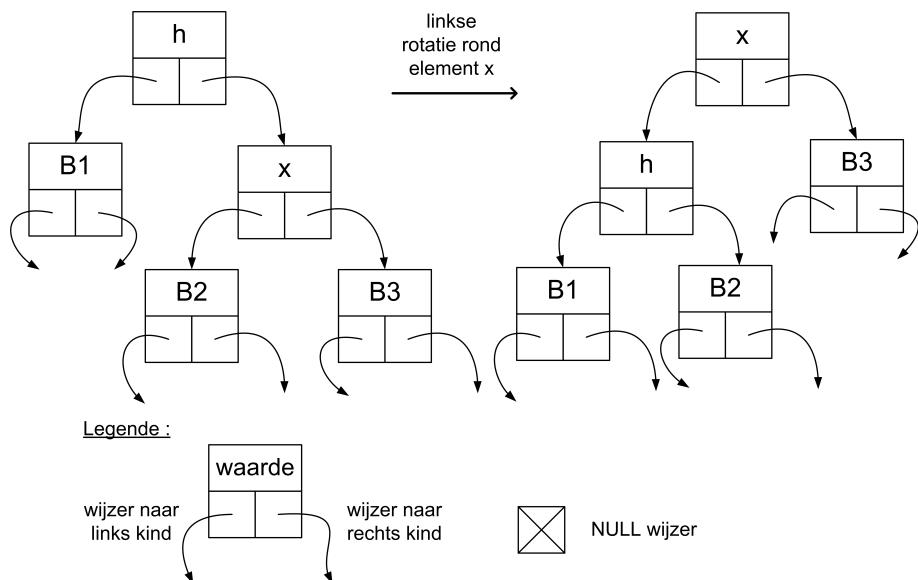
Het is belangrijk om in te zien dat een nieuw element niet zomaar bovenaan kan toegevoegd worden. Daarom wordt de toevoeging onderaan gedaan, en aan de hand van *rotaties* wordt het nieuwe element gepromoveerd naar de wortel van de boom. Rotaties komen aan bod in volgende sectie.

Rotatie in binaire zoekbomen - rotation

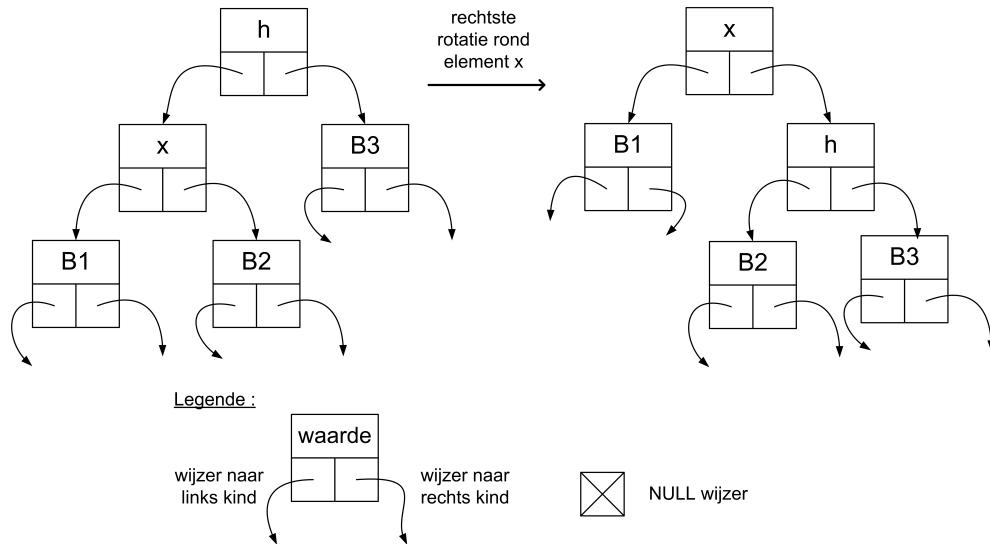
Een rotatie is een lokale transformatie op een deel van een binaire zoekboom die toelaat om de rol van de wortel en één van zijn kinderen om te wisselen, en dit terwijl de voorwaarden tussen de knopen van een binaire zoekboom behouden blijven.

Rotatie is een basisoperatie die gebruikt wordt om de knopen in een binaire zoekboom te herschikken. Zoals hierboven vermeld wordt bij toevoegen van een element in de wortel (Eng.: root insertion) het element onderaan in de binaire zoekboom toegevoegd en dan de boom herschikt tot de nieuwe knoop in de wortel staat. Rotaties worden zeer dikwijls **recursief** toegepast.

Figuur 2.7 toont een linker-rotatie waarbij element x naar de wortel gepromoveerd wordt, terwijl in figuur 2.8 een rechter-rotatie getoond wordt.



Figuur 2.7: Linker-rotatie in een binaire zoekboom.



Figuur 2.8: Rechter-rotatie in een binaire zoekboom.

Element verwijderen uit een binaire zoekboom - remove

Een element kan niet zomaar uit een binaire zoekboom verwijderd worden (omdat de voorwaarden voor een binaire zoekboom dan kunnen geschonden worden). De beste manier is de volgende:

- promoveer in de rechter sub-boom het kleinste element (i.e. meest linkse) naar de root van deze sub-boom,
- dit element krijgt dezelfde linker sub-boom als het verwijderde element,
- dit element komt in de volledige boom in de plaats van het verwijderde element.

Binaire zoekbomen samenvoegen - join

Beschouwen we twee binaire zoekbomen: BST_1 en BST_2 . Een mogelijke implementatie om BST_1 en BST_2 samen te voegen is elk knooppunt van BST_1 te overlopen en toe te voegen aan BST_2 . Dit is een zeer eenvoudige implementatie, maar kan efficiënter op de volgende manier:

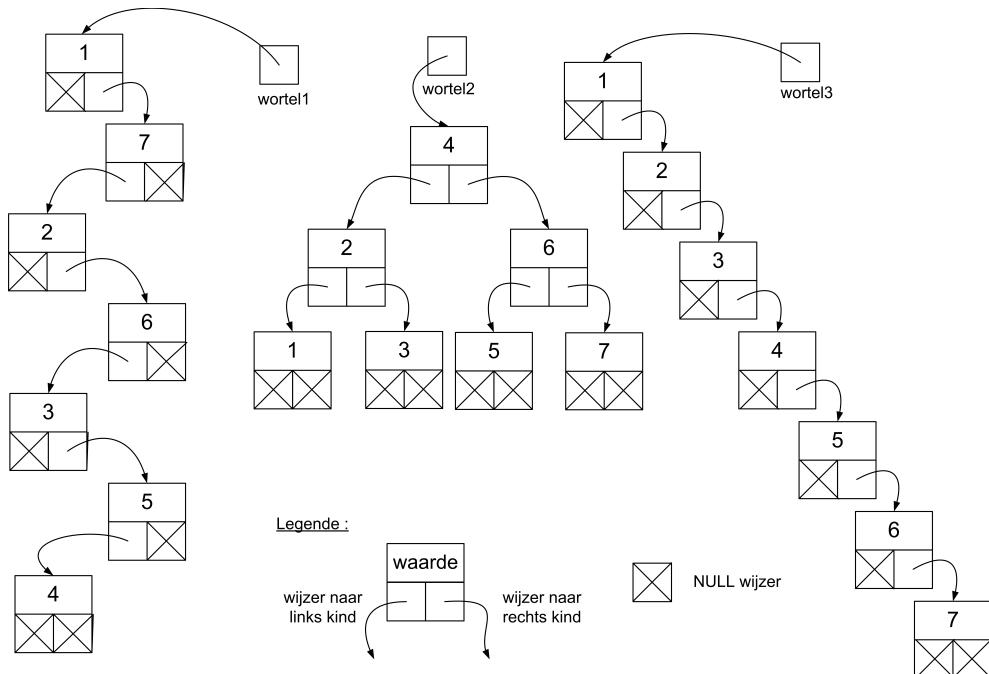
- Voeg de wortel van BST_1 bij BST_2 aan de hand van root insertion (cfr. sectie 2.3.3).
- Dit geeft 2 sub-bomen (linker sub-bomen van BST_1 en BST_2) die kleiner zijn dan de nieuwe root in BST_2 en 2 sub-bomen (rechter sub-bomen van BST_1 en BST_2) die groter zijn dan de nieuwe root in BST_2 .
- We passen de samenvoeging recursief toe op de respectieve sub-bomen.

Iedere node komt hierbij hoogstens éénmaal aan bod, hetgeen een efficiënte implementatie oplevert.

Prestatiekenmerken van binaire zoekbomen

De prestatie van binaire zoekbomen is vergelijkbaar met de prestatie van Quicksort. De ouder-knoop fungeert telkens als partitie-element. Net als bij Quicksort zijn de prestaties afhankelijk van de volgorde waarin de elementen zijn toegevoegd. Deze volgorde bepaalt bij een binaire zoekboom de vorm van de boom. Voor een gegeven verzameling van elementen, bestaan er dus meerdere equivalente binaire zoekbomen.

Figuur 2.9 toont drie equivalente binaire zoekbomen (m.a.w. ze bevatten dezelfde informatie): de middelste boom is de meest efficiënte (drie niveau's), terwijl de linkse en rechtse boom het maximaal aantal niveau's bevatten (rechterboom wordt een lineaire boom genoemd, terwijl de linkerboom een zig-zag boom genoemd wordt, wegens alternende volgorde van de elementen). De middelste binaire zoekboom wordt een **perfect gebalanceerde boom** genoemd.



Figuur 2.9: Drie equivalente binaire zoekbomen: (links) zig-zag boom, (midden) perfect gebalanceerde boom, (rechts) lineaire boom.

Lineaire bomen of zig-zag bomen vertonen de slechtste prestatie voor zoeken en toevoegen van elementen: proportioneel met N , het aantal elementen in de binaire zoekboom. Een perfect gebalanceerde boom vertoont de beste prestatie voor zoeken en invoegen van elementen: proportioneel met $\log_2(N)$.

Indien L het aantal niveau's in de boom voorstelt en N het aantal elementen, kan men

volgende uitdrukking afleiden:

$$1 + 2 + \dots + 2^{(L-2)} + 2^{(L-1)} \geq N$$

of:

$$2^L - 1 \geq N.$$

Hieruit kan gemakkelijk volgende uitdrukking voor L afgeleid worden:

$$\log_2(N + 1) \leq L \leq N.$$

Een perfect gebalanceerde boom heeft dezelfde prestatie als een gesorteerde array, waarop het binair zoeken (Eng.: binary search) algoritme toegepast wordt. Dit binair zoeken algoritme is ook een voorbeeld van een verdeel en heers algoritme:

- Verdeel de gesorteerde array in twee en bepaal in welk deel het element ligt
- Ga verder met die helft

De prestatie om een element te zoeken is dezelfde als bij een binaire zoekboom, het invoegen van een nieuw element in de array is echter veel minder efficiënt (aantal operaties proportioneel met N).

2.3.4 Gebalanceerde zoekbomen - balanced search trees

Zoals hierboven (sectie 2.3.3) vermeld, zijn binaire zoekbomen het meest efficiënt als ze perfect gebalanceerd zijn. Daarom probeert men bij het invoegen van elementen de boom zo perfect mogelijk te balanceren. In deze sectie worden enkele veelgebruikte methoden vermeld om deze balansering te bereiken.

Zoekbomen met toevallige verdeling - randomized trees

Om te vermijden dat toevoegen van reeds gesorteerde elementen een lineaire deel-boom oplevert, voert men een kans in dat een nieuw element de nieuwe wortel wordt. Als er een nieuwe knoop toegevoegd wordt aan de binaire zoekboom, is er een kans $\frac{1}{N+1}$ dat deze de nieuw wortel (Eng.: root) wordt van de sub-boom. Iedere knoop dient hiervoor bij te houden hoeveel knopen er in de sub-boom zijn, bijvoorbeeld een extra veld met naam N . Het root insertion algoritme wordt recursief opgeroepen en tijdens elke oproep wordt de kans opnieuw berekend voor de beschouwde sub-boom.

Verwijde zoekbomen - splay trees

Doelstelling van verwijde zoekbomen is een boomstructuur opbouwen die zichzelf (zo optimaal mogelijk) reorganiseert na iedere toevoeging van een element. Hiertoe worden **dubbele rotaties** uitgevoerd, waarbij een kleinkind in de wortel van de sub-boom geplaatst wordt. Er worden twee gevallen onderscheiden afhankelijk van de oriëntatie van de verbindingen tussen de knopen: *zig-zig* rotatie en *zig-zag* rotatie. Deze rotaties vallen buiten het bestek van deze cursus. Men kan aantonen dat toepassing van deze rotaties meer gebalanceerde zoekbomen opleveren dan toepassing van de enkelvoudige rotatie (figuren 2.7 en 2.8) en zoekbomen met toevallige verdeling.

2-3-4 bomen

In dit geval worden 1, 2 of 3 informatie-elementen per knooppunt toegelaten, men onderscheidt drie types knooppunten:

- 2-knooppunt: één informatie-element, twee kinderen,
- 3-knooppunt: twee informatie-elementen, drie kinderen,
- 4-knooppunt: drie informatie-elementen, vier kinderen.

In geval van meerdere informatie-elementen per knooppunt (bij 3-knooppunten en 4-knooppunten), zijn deze informatie-element steeds van klein naar groot gesorteerd.

Bij een 2-knooppunt zijn linkse kinderen kleiner dan het informatie-element en de rechtse kinderen groter of gelijk aan het informatie-element. Bij een 3-knooppunt, zijn de linkerkinderen kleiner dan het eerste informatie-element, de middelste kinderen zijn groter of gelijk aan het eerste informatie-element en kleiner dan het tweede informatie-element, de rechterkinderen zijn groter dan of gelijk aan het tweede informatie-element. Bij een 4-knooppunt, zijn de linkerkinderen kleiner dan het eerste informatie-element, de elementen van de tweede kinderboom zijn groter of gelijk aan het eerste informatie-element en kleiner dan het tweede informatie-element, de elementen van de derde kinderboom zijn groter of gelijk aan het tweede informatie-element en kleiner dan het derde informatie-element, de rechterkinderen zijn groter dan of gelijk aan het derde informatie-element.

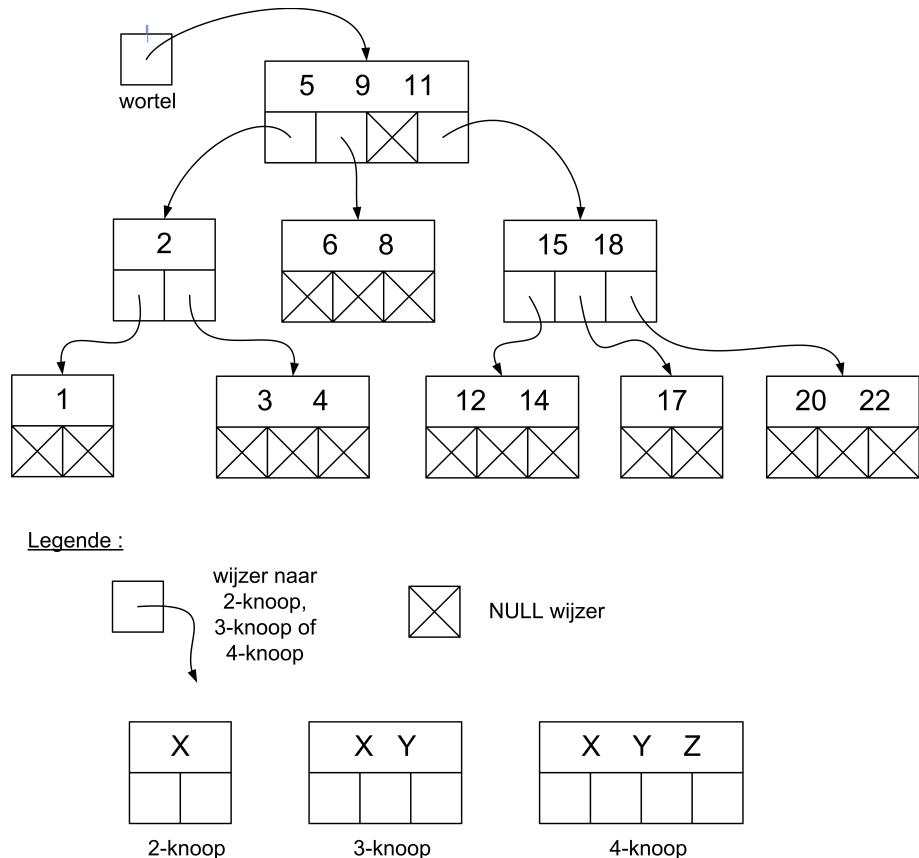
Figuur 2.10 toont een voorbeeld van een 2-3-4 boom, die gehele getallen bevat.

Het zoeken van een element in 2-3-4 boom gebeurt vergelijkbaar met binaire zoekbomen: de waarden in het knooppunt worden vergeleken met het te zoeken element, en het interval waarin dit ligt bepaalt welke sub-boom verder beschouwd wordt.

Het toevoegen van een element is ook vergelijkbaar met binaire zoekbomen: er wordt tot in de toppen van de boom gezocht naar het element, bij een *search miss*, wordt één van de volgende regel toegepast:

- indien de *search miss* bij een 2-knooppunt optreedt: transformeer deze naar een 3-knooppunt,
- indien de *search miss* bij een 3-knooppunt optreedt: transformeer deze naar een 4-knooppunt,
- indien de *search miss* bij een 4-knooppunt optreedt: transformeer deze, waardoor ook een informatie-element in het ouder-knooppunt bijgevoegd wordt. Deze transformaties vallen buiten het bestek van deze cursus.

2-3-4 bomen groeien (m.a.w. er komt een extra niveau bij) als de wortel gesplitst wordt. Dit type van datastructuur vergt een complexere implementatie dan binaire zoekbomen, maar het grote voordeel is dat de balans altijd behouden wordt (steeds perfecte balancering) onafhankelijk van het aantal toevoegingen of opzoeken.



Legende :

wijzer naar 2-knoop,
3-knoop of
4-knoop

NULL wijzer

X
2-knoop

X Y
3-knoop

X Y Z
4-knoop

Figuur 2.10: Voorbeeld van een 2-3-4 boom, die gehele getallen bevat.

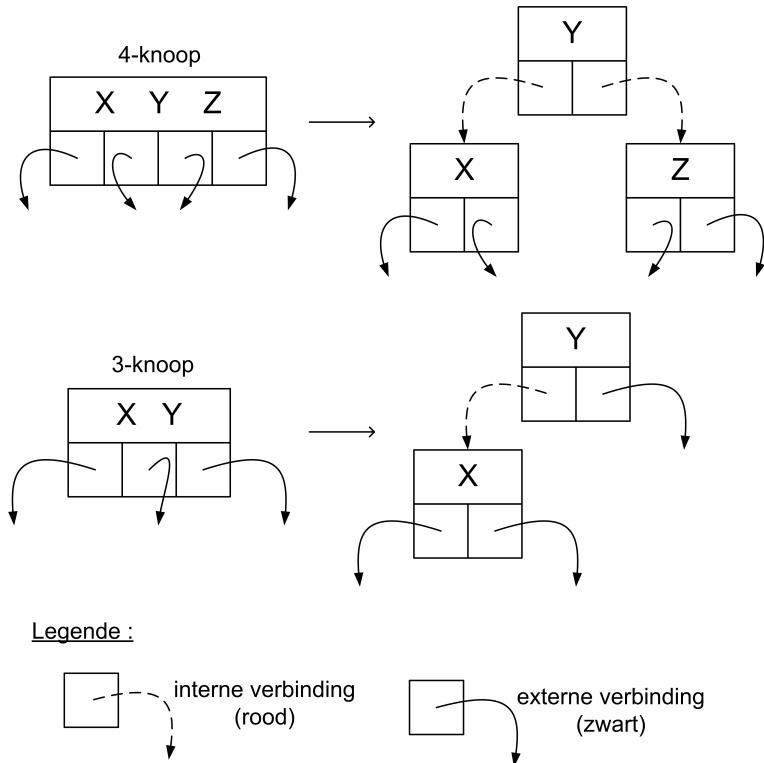
Rood-zwart bomen - red-black trees

Rood-zwart bomen zijn gebaseerd op de principes van 2-3-4 bomen, maar geïmplementeerd aan de hand van binaire zoekbomen, waarbij de randlijnen een kleur krijgen (extra veld bij elke verbinding):

- interne randlijnen in het rood: geven aan welke knooppunten samenhoren en samen een 3-knoop of 4-knoop vormen.
- externe randlijnen in het zwart: geven de verbindingen aan tussen de 2-, 3- en 4-knooppunten.

Figuur 2.11 toont het verband tussen de knooppunten van 2-3-4 bomen en de knooppunten van rood-zwart bomen.

De standaard zoekmethodes voor binaire zoekbomen werken zonder wijziging. Om de balans en de voorwaarden te behouden tijdens het invoegen van elementen dienen eenvoudige operaties (bijv. kleuren wisselen, enkelvoudige rotaties, etc.) toegepast te worden: deze komen in deze cursus echter niet in detail aan bod en zijn geen onderdeel



Figuur 2.11: Verband tussen de knooppunten van 2-3-4 bomen en de knooppunten van rood-zwart bomen.

van de leerstof.

Door het feit dat de balans telkens perfect behouden wordt, zijn rood-zwart bomen efficiënter dan verwijde zoekbomen (sectie 2.3.4). Er zijn ook minder rotaties nodig dan met verwijde bomen.

Omwille van deze redenen, worden rood-zwart bomen zeer veel in de praktijk gebruikt. Zo worden ze gebruikt in de implementatie van de C++ STL types: `map`, `multimap` en `multiset` en de Java types: `TreeMap` en `TreeSet`.

2.4 Hopen - Heaps

2.4.1 Wachtlijnen met prioriteit - priority queues

Wachtlijnen met prioriteit bevatten gegevens, waarbij met elk informatie-element een bepaalde prioriteit geassocieerd is. Deze datastructuur biedt een operatie *getmax* aan om zeer snel het element met de hoogste prioriteit te kunnen teruggeven en te verwijderen uit de wachtrij. De waarde van de informatie-elementen zelf kan ook als prioriteit aanzien worden en de *getmax* operatie levert dan het informatie-element met de grootste waarde en verwijdert deze uit de wachtrij.

Een mogelijke implementatie van een wachtrij met prioriteit is aan de hand van een gesorteerde array, waarbij het grootste element snel kan gevonden worden: het sorteren van array's en verwijderen van elementen is echter niet zo performant (proportioneel met N in het beste geval). Een efficiëntere implementatie wordt in de volgende sectie besproken.

2.4.2 Heap datastructuur

Een heap datastructuur is een *complete binaire boom* T zodat:

- T leeg is, of
- het informatie-element van het ouderknooppunt van T is groter dan of gelijk aan de informatie-elementen van zijn kinderen, en de sub-bomen van dit knooppunt zijn op hun beurt heap's.

De ouderknoop van de heap bevat het element met de grootste waarde. Er is geen verband tussen de waarden van de informatie-elementen van de kinderen van een bepaald knooppunt.

Bij een complete binaire boom worden de aanwezige plaatsen in de boom zo goed mogelijk van links naar rechts opgevuld: de boom is dan gebalanceerd en enkel op het laagste niveau aan de rechterkant zijn er mogelijk lege plaatsen (bij toevoegen van een element worden deze dan van links naar rechts opgevuld, tot wanneer het huidige niveau volzet is en op het volgende niveau weer volledig links een nieuw element toegevoegd wordt). Deze eigenschap van complete binaire bomen, zorgt ervoor dat een heap datastructuur ook aan de hand van een array kan voorgesteld worden. De array index wordt dan gebruikt om de boom te doorkruisen:

- de voorvader van het knooppunt k bevindt zich op de plaats $\frac{k}{2}$ in array,
- de kinderen van het knooppunt k bevinden zich op de posities $2 \times k$ en $2 \times k + 1$ in de array.

Figuur 2.12 toont een voorbeeld van een heap datastructuur en de corresponderende array-voorstelling.

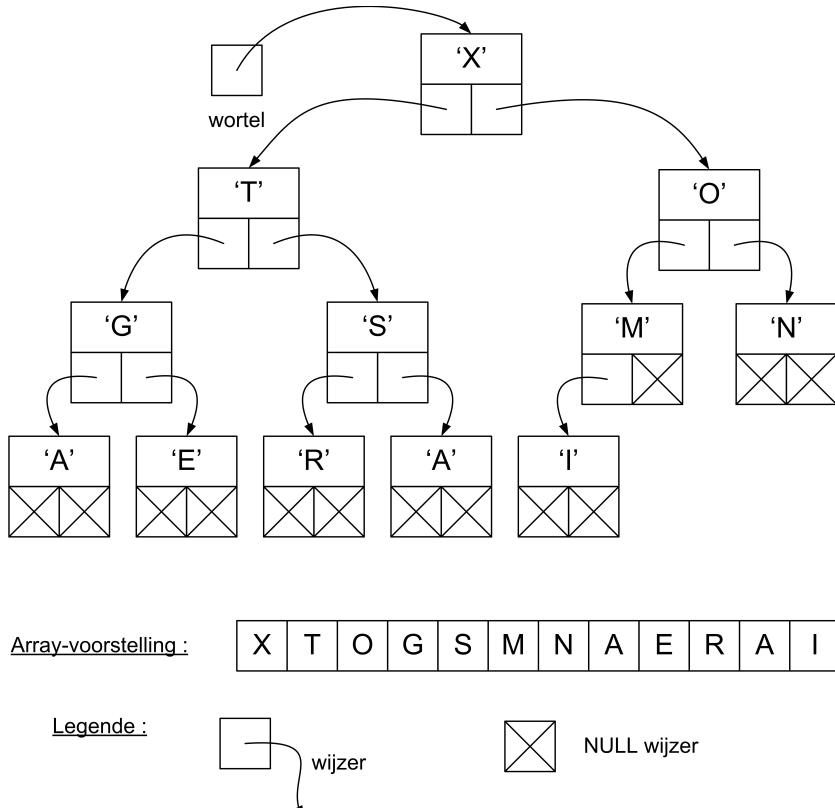
2.4.3 Heap algoritmen

Promotie of herstel van beneden naar boven - fixup

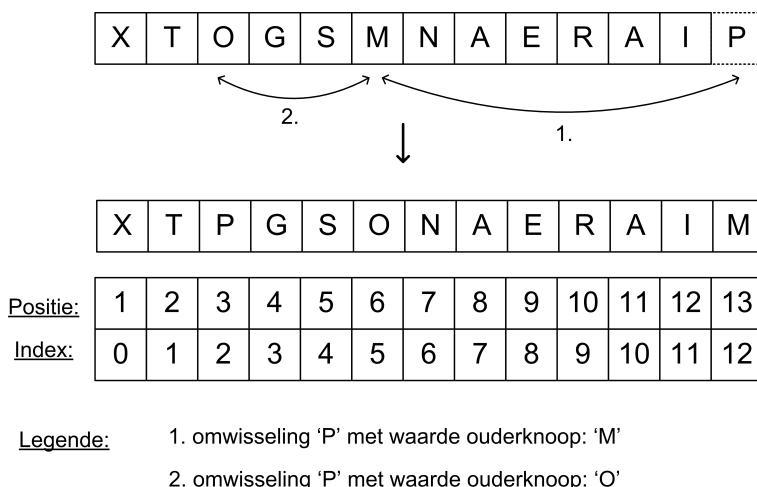
Wanneer een knooppunt een grotere waarde heeft dan zijn voorvader, wordt de heap-voorwaarde hersteld door:

- omwisseling met de voorvader,
- ga omhoog in de boom,
- ga door tot een voorvader een grotere waarde heeft dan zijn kinderen.

Dit wordt het fixup algoritme genoemd en wordt geïllustreerd in figuur 2.13.



Figuur 2.12: Voorbeeld van een heap datastructuur, die karakters als elementen bevat. De corresponderende array-voorstelling wordt eveneens getoond.



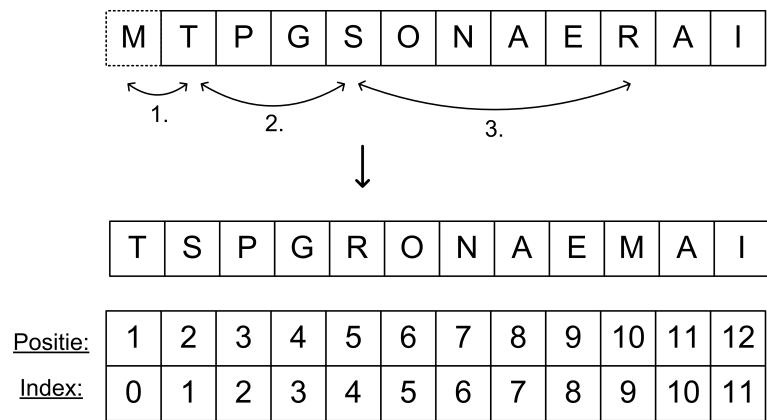
Figuur 2.13: Illustratie van het fixup algoritme voor heap datastructuren.

Degradatie of herstel van boven naar beneden - fixdown

Wanneer een knooppunt een kleinere waarde heeft dan zijn kinderen, wordt de heap voorwaarde hersteld door:

- omwisseling met het grootste kind,
- zak verder af in de boom,
- ga door tot de voorvader een grotere waarde heeft dan de kinderen.

Dit wordt het fixdown algoritme genoemd en wordt geïllustreerd in figuur 2.14.



- Legende:
1. omwisseling 'M' met waarde grootste kind: 'T'
 2. omwisseling 'M' met waarde grootste kind: 'S'
 3. omwisseling 'M' met waarde grootste kind: 'R'

Figuur 2.14: Illustratie van het fixdown algoritme voor heap datastructuren.

Element toevoegen - insert

Een element toevoegen gebeurt achteraan en vervolgens wordt het fixup algoritme toegepast op dit nieuwe element.

Verwijderen grootste element - getmax

Het grootste element verwijderen gebeurt door het eerste element te selecteren, het laatste element vooraan te plaatsen en vervolgens het fixdown algoritme toe te passen op dit element.

2.4.4 HeapSort

Sorteren van een array kan gebeuren aan de hand van een heap datastructuur: de elementen worden er achtereenvolgens in geplaatst en nadien wordt telkens het grootste

element verwijderd en in volgorde opgeslagen. Deze implementatie vereist extra geheugen om de heap datastructuur op te slaan. Men kan echter ook rechtstreeks op de oorspronkelijke array de heap principes toepassen:

- op het voorlaatste niveau van de heap wordt het fixdown algoritme toegepast op het ouderknooppunt van alle sub-bomen (die uit 1, 2 of 3 elementen bestaan),
- dit wordt herhaald op het bovenliggende niveau, enz. (tot wanneer de heap voorwaarde in de array voldaan is),
- het grootste element (voorste element in de array) wordt omgewisseld met het laatste element,
- op dit nieuwe voorste element wordt het fixdown algoritme toegepast,
- dit wordt herhaald op de sub-array, i.e. zonder het laatste element,
- tot wanneer de sub-array slechts twee elementen bevat, die door omwisseling gesorteerd worden.

Heapsort vereist in het slechtste geval $N \times \log_2(N)$ operaties en geen extra geheugen. Het is daarom een zeer efficiënte sorteermethode voor array's, die heel vaak gebruikt wordt.

2.5 Grafen - Graphs

2.5.1 Definitie

Een graaf is een verzameling knooppunten (Eng.: vertices) en een verzameling verbindingen tussen deze knooppunten (Eng.: edges) met of zonder richting (Eng.: directed - undirected).

2.5.2 Voorstelling van grafen

Matrix-voorstelling

Er wordt een vierkante matrix \mathbf{g} bijgehouden met evenveel kolommen en rijen als er knooppunten in de graaf zijn. Als we het aantal knooppunten in de graaf voorstellen door N , dan geldt ($i=0\dots N-1$, $j=0\dots N-1$):

$$g[i][j] = 0 \text{ indien er geen verbinding is tussen knooppunt } i \text{ en } j,$$

en

$$g[i][j] = 1 \text{ indien er wel verbinding is tussen knooppunt } i \text{ en } j.$$

De matrix is dus symmetrisch wanneer alle verbindingen zonder richting zijn of wanneer alle verbindingen bi-directioneel zijn.

Lijst-voorstelling

Er wordt voor elk knooppunt een lineaire lijst bijgehouden waarin alle knooppunten opgeslagen zijn, die rechtstreeks vanuit het beschouwde knooppunt kunnen bereikt worden (de buur-knooppunten). Deze kunnen in een willekeurige volgorde in de lijst opgeslagen worden of bijvoorbeeld alfabetisch gesorteerd volgens de naam van het knooppunt.

2.5.3 Doorlopen van grafen

Er zijn twee manieren om de knooppunten van een graaf te doorlopen.

Diepte doorkruising - depth first

Er wordt gestart in een bepaald knooppunt, de eerste buur wordt doorlopen, en vervolgens de eerste buur van deze buur, enz, tot wanneer een reeds doorlopen knooppunt tegengekomen wordt. Als dit laatste het geval is, wordt de volgende buur van het laatst beschouwde knooppunt doorlopen. Dit proces gaat verder tot wanneer de volledige graaf doorlopen is.

Breedte doorkruising - breadth first

In dit geval worden eerst alle buurknooppunten van het start-knooppunt doorlopen, en vervolgens alle buurknooppunten van deze buurknooppunten. Er dient ook telkens bijgehouden worden welke knooppunten reeds doorlopen zijn, zodat elk knooppunt slechts éénmaal doorlopen wordt.

2.6 Hashtabellen - Hashtables

2.6.1 Sleuteltransformaties - hashing

Een hashtabel bestaat uit een array, waarin informatie opgeslagen wordt aan de hand van sleutel-waarde paren (Eng.: key-value pairs). Op basis van sleutel wordt de index in de array berekend. M stelt hierbij de grootte van de array voor en N het effectief aantal opgeslagen elementen. De berekening van de index wordt sleuteltransformatie (Eng.: hashing) genoemd: de waarde van de sleutel wordt getransformeerd naar een index in het bereik $0 \dots M - 1$. In bepaalde gevallen bevatten de sleutels de nodige informatie en worden geen sleutel-waarde paren opgeslagen, maar enkel de sleutels.

Het is mogelijk dat twee verschillende sleutels dezelfde index opleveren: dit wordt een botsing (Eng.: collision) genoemd en de datastructuur en bijhorende algoritmen zorgen voor de oplossing hiervan (sectie 2.6.3).

2.6.2 Sleuteltransformatiefuncties - hash functions

Deze functies zorgen voor herleiding van de sleutels naar een gehele waarde in het bereik: $0 \dots M - 1$. Ingeval van gehele sleutels wordt dikwijls de $\%M$ -functie (rest bij deling door

M) genomen, waarbij M een priemgetal is. Andere hash-functies kunnen ook bedacht worden: complexere hash-functies zijn echter geen onderdeel van de leerstof.

Doelstelling van hashfuncties is steeds om de sleutels willekeurig en uniform over de array te kunnen verspreiden (geen clustering), zodat de probabilititeit voor iedere positie in de tabel gelijk is voor alle sleutels.

2.6.3 Oplossen van indexconflicten

Twee technieken bestaan voor oplossing van index conflicten, ook wel botsingen (Eng.: collisions) genoemd. Deze twee technieken komen hieronder aan bod.

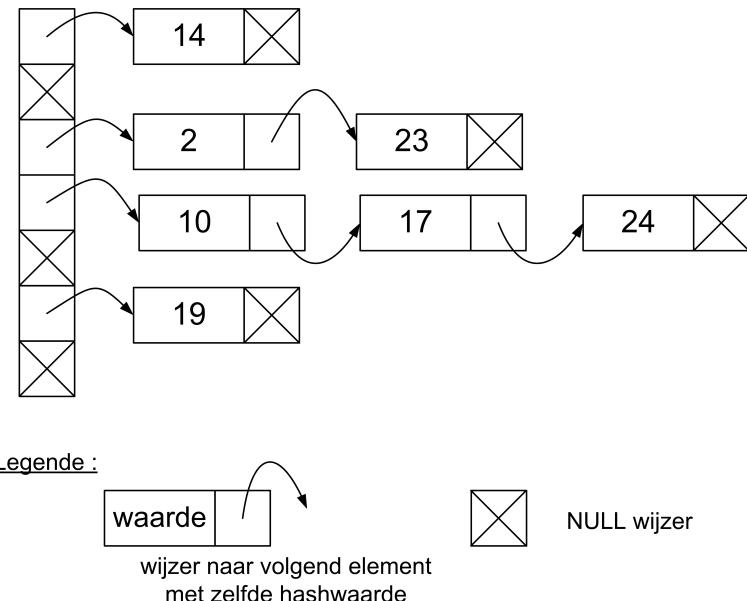
Afzonderlijk geschakelde lijsten - separate chaining

Bij deze techniek worden sleutels die botsen in een geschakelde lijst geplaatst. De geschakelde lijst is geassocieerd met de berekende index. Figuur 2.15 illustreert een hashtabel met afzonderlijk geschakelde lijsten, die gehele getallen als waarden bevat. Het zoeken van een element vereist het doorlopen van de geschakelde lijst. Men kan de elementen gesorteerd of ongesorteerd (bijvoorbeeld telkens vooraan of achteraan) in de lijst plaatsen. Gesorteerd invoegen vergt meer berekeningen bij het invoegen, maar het opzoeken van een element gaat sneller (afweging te maken bij de implementatie van de hashtabel). Vermits men de geschakelde lijsten linear moet doorlopen, is het niet aangewezen dat elke lijst zeer veel elementen bevat. Als vuistregel kiest men $M \geq \frac{N}{10}$, zodat de gemiddelde lengte van een geschakelde lijst maximaal 10 is. Gemiddeld zijn er dan $\frac{N}{M}$ sleutels per positie in de tabel.

Open adres schema - open address scheme

Deze techniek bestaat erin dat als twee sleutels botsen (i.e. zelfde index opleveren) er een alternatieve (niet bezette) positie in de tabel gekozen wordt. Idealiter zijn er veel lege posities in de array, zodat de ganse array niet hoeft overlopen te worden om een lege positie te vinden. Als vuistregel kiest men dikwijls $M \geq N \times 2$, zodat maximaal de helft van de elementen van de hashtabel bezet is. Vermits het adres van een element (i.e. de uiteindelijk gekozen index) afhangt van de hash-waarde, maar ook van de reeds bezette posities in de array, kan dit adres in principe elke mogelijke index-waarde aannemen (vandaar de benaming *open*).

Lineaire probes - linear probing In geval van een index-conflict wordt de index steeds met 1 verkleind, tot wanneer er een lege positie gevonden wordt. Dit wordt geïllustreerd in figuur 2.16. Een nadeel van deze aanpak is dat de opvulling van de posities lokaal gebeurt en dus lokaal lege elementen opgevuld worden. Dit geeft aanleiding tot groepering (clustering) van elementen, hetgeen een slechte invloed heeft op de prestaties (het duurt langer eer een lege positie gevonden wordt en het opzoeken van een element duurt ook langer vermits er gezocht dient te worden tot wanneer men een lege positie tegenkomt of



Figuur 2.15: Illustratie van een hashtabel met afzonderlijk geschakelde lijsten, die gehele getallen als waarden bevat ($M=7$). Als hash-functie wordt $(.) \% 7$ (rest bij deling door 7) genomen.

het element gevonden wordt). Daarom wordt vaak een dubbele transformatie gebruikt, zoals hieronder uitgelegd.

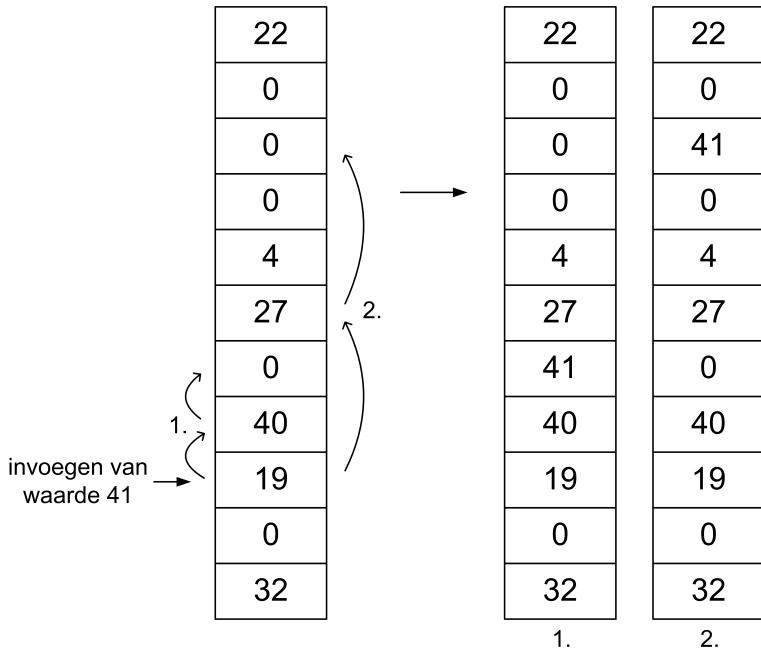
Dubbele transformatie functies - double hashing In dit geval wordt de sleutel vertaald door een tweede hash-functie, die de sprongwaarde in de array berekent. Deze sprongwaarde wordt gebruikt bij index-conflicten, om met de berekende sprong-waarde door de array te lopen, tot wanneer een lege positie gevonden wordt. De sprongwaarden zijn verschillend voor de sleutels die conflicteren.

Dit wordt ook geïllustreerd in figuur 2.16. Een voordeel van deze aanpak is dat groeperingen (clusters) vermeden worden, hetgeen dus de prestaties van de hashtabel ten goede komt.

Een nadeel van open adres schema's is dat het verwijderen van elementen een complexe operatie is.

2.6.4 Herschalen van de hashtabel - rehashing

Wanneer de hashtabel teveel elementen bevat (cfr de uitdrukkingen hierboven met verband M en N), is het nodig om deze te herschalen: de lengte van de array kan bijvoorbeeld verdubbeld worden, zodat er voldoende ruimte is om de volgende elementen toe te voegen. Let wel: een aanpassing van M vereist de herberekening van de indices en



Legende : 1. Invoegen waarde 41 door lineaire probing

 2. Invoegen waarde 41 door dubbele hashing

Figuur 2.16: Illustratie van een hashtabel met open adres schema, die gehele getallen bevat ($M=11$). De waarde nul in de array duidt op een lege positie. Bij invoeging van waarde 41, vindt er een botsing (Eng.: collision) plaats, die via 1. lineaire probing of 2. dubbele hashing wordt opgelost. Als tweede hashfunctie wordt $(.) \bmod 11$ (gehele deling door 11) genomen.

de herplaatsing van de elementen in de tabel (hetgeen door de term *rehashing* wordt aangeduid)!

2.7 Gecombineerde datastructuren - combined datastructures

De datastructuren in dit hoofdstuk kunnen uiteraard gecombineerd worden tot de meest aangewezen datastructuur voor de beoogde applicatie. Enkele voorbeelden van combinaties:

- een hashtabel kan wijzers naar bomen bevatten: de hash-waarde van de sleutel bepaalt dan in welke boom de elementen opgeslagen worden,
- een binaire zoekboom kan array's als elementen bevatten, bijvoorbeeld om bijéénhorende elementen samen in een array op te slaan,

- een gelinkte lijst kan wijzers naar heap's als elementen bevatten, waarbij bijvoorbeeld de gelinkte lijst gesorteerd is op de grootste elementen van de heap's.

Deel II

Software Ontwikkeling in C

Hoofdstuk 3

Situering Programmeertaal C

3.1 Korte historiek

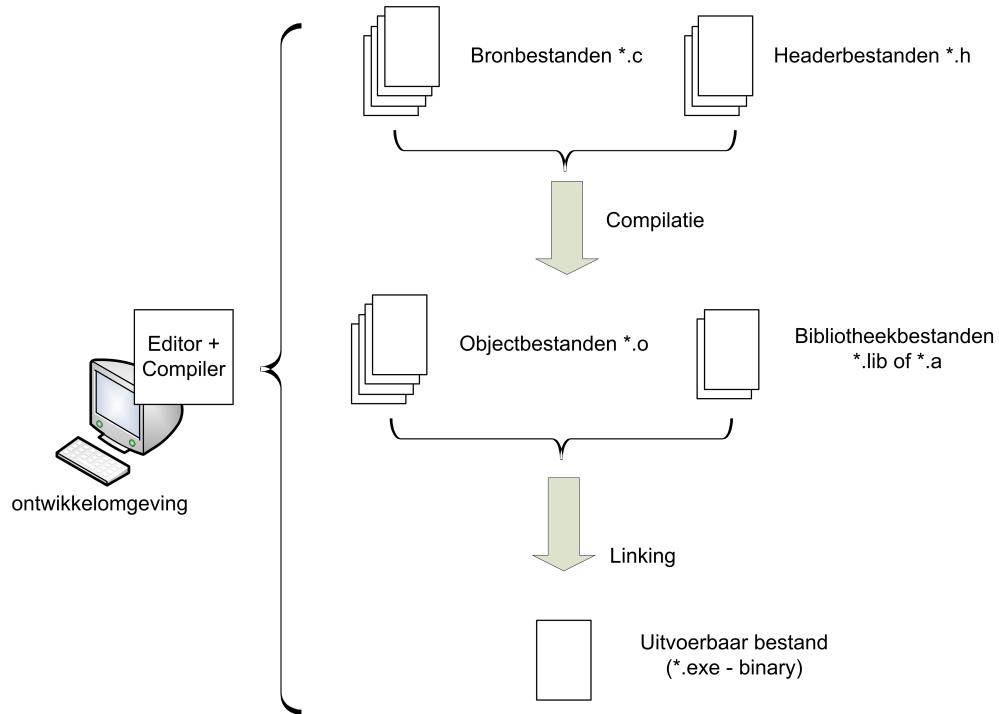
De programmeertaal C is in 1972 in de AT&T Bell Labs (New Jersey, Verenigde Staten) ontwikkeld door Dennis Ritchie en Ken Thompson. De naam C werd gekozen vermits de taal aanzien wordt als de opvolger van de programmeertaal B (ontwikkeld door Ken Thompson) en de programmeertaal A. Het primaire doel van de taal C was het programmeren van besturingssystemen (lees: in 1972 het programmeren van Unix) en is vooral bedoeld voor professionele programmeurs en stelt laagniveau-primitieven beschikbaar aan de programmeur (noodzakelijk vermits het primaire doel de ontwikkeling van besturingssystemen was). De taal is geoptimaliseerd voor snelle uitvoering (in tegenstelling tot de programmeertaal Java) en laat soms cryptische constructies toe.

Er bestaan verschillende varianten van C, de meest gekende is ANSI C. Dit is de variant die door de American National Standards Institute (ANSI) gestandaardiseerd werd en wereldwijd als de referentie aanzien wordt. In deze cursus gaat onze aandacht dan ook uit naar ANSI C.

3.2 Interpretatie versus compilatie

Java is een voorbeeld van een geïnterpreteerde taal, waarbij de vertaling in machinetaal gebeurt tijdens de uitvoering van het programma. Het grote voordeel van interpretatie is de porteerbareheid van de code. Een nadeel is echter de uitvoeringssnelheid vermits de interpreter zorgt voor een extra vertraging in het uitvoeren van de code. Bij een gecompileerde taal (zoals C) wordt de vertaling in machinetaal gedaan tijdens de ontwikkeling van het programma, als een deel van het ontwikkelproces. Compilatie komt de uitvoeringssnelheid ten goede maar heeft als nadeel de lagere porteerbareheid. Een gecompileerd programma voor één processortype dient in de meeste gevallen gehercompileerd te worden voor een ander processortype.

Figuur 3.1 toont schematisch de verschillende stappen om in C via compilatie en linking bronbestanden (Eng.: source files) om te zetten naar een uitvoerbaar bestand (Eng.: executable file of ook *binary file* genoemd).



Figuur 3.1: Principe van compilatie en linking in C

Zowel compilatie als linking komen verder in hoofdstuk 5 uitgebreider aan bod.

3.3 Uitvoeringssnelheid versus ontwikkelsnelheid en ontwikkelgemak

Java-programma's zijn doorgaans gemakkelijker te ontwikkelen dan C-programma's omdat Java zorgt voor automatisch dynamisch geheugenbeheer (dankzij de garbage collector) en het automatisch gooien van run-time exceptions wanneer een illegale operatie uitgevoerd wordt. Voorbeelden van illegale operaties zijn het oproepen van een methode op een NULL referentie of de toegang tot een element buiten de grenzen van een array. In C is het dynamisch geheugenbeheer volledig onder controle van de ontwikkelaar en worden geen automatische controles voor illegale operaties uitgevoerd.

Samenvattend: de uitvoeringssnelheid van C-programma's is doorgaans hoger dan bij Java-programma's, maar het schrijven van C-programma's vereist een grotere verantwoordelijkheid van de ontwikkelaar.

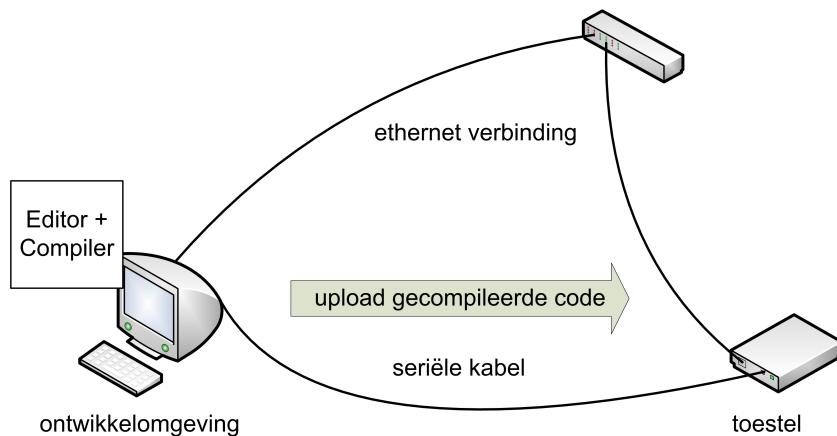
3.4 Situering toepasbaarheid

Er wordt dikwijls onderscheid gemaakt tussen algemeen toepasbare software (Eng.: general purpose software) en software met gerichte toepasbaarheid (Eng.: software for dedicated systems). Voorbeelden van de eerste categorie zijn software systemen voor

banken, administratie, data-analyse, en voorbeelden van de tweede categorie zijn software systemen voor domotica, sensoren, vliegtuigen, netwerkapparatuur, plotters, camera's, besturingssystemen voor mobiele telefoons, etc.

Voor de eerste categorie zijn Java en .Net momenteel veruit de meest gebruikte software technologieën, voor de tweede categorie worden uitsluitend C en in sommige gevallen C++ gebruikt.

Indien software met gerichte toepasbaarheid ontwikkeld wordt voor uitvoering op een specifiek toestel, dient men het principe van *cross-compilatie* toe te passen: de applicatie wordt ontwikkeld op een desktop of laptop en daar gecompileerd voor uitvoering op het specifieke toestel (i.e. de compiler vertaalt de broncode naar instructies voor de CPU van het toestel, die dus door het toestel kunnen uitgevoerd worden). Figuur 3.2 toont schematisch het principe van cross-compilatie.



Figuur 3.2: Principe van cross-compilatie voor ontwikkeling van software met gerichte toepasbaarheid

3.5 Programmeeromgeving

Tijdens de oefeningenlessen zal gewerkt worden met Microsoft Visual C++ Express, dat in de PC-klassen geïnstalleerd is. Deze programmeeromgeving kan door de studenten ook gratis gedownload worden van de Microsoft website. Via [athena](#) kan de programmeeromgeving ook online gebruikt worden. Details over het gebruik van deze programmeeromgeving zullen tijdens de oefeningenlessen aangeleerd worden.

3.6 Belangrijk

Vermits C veel vrijheid biedt aan de programmeur en minder faciliteiten biedt dan Java om fouten tijdens het compileren of uitvoeren op te sporen, is het aangeraden om programmeer-discipline aan de dag te leggen.

Hoofdstuk 4

C Taalelementen

In dit hoofdstuk worden de basiselementen van C behandeld. Na dit hoofdstuk is de student in staat eenvoudige C programma's te begrijpen en zelf te coderen, met volgende onderdelen: definitie van variabelen, type conversies, gebruik van operatoren, definitie en gebruik van structs en unions, input en output en gebruik wiskundige functies.

4.1 Syntaxregels en taalelementen

In het bijhorende handboek wordt de syntax van C telkens formeel vastgelegd aan de hand van BNF. Hieronder wordt het principe van BNF behandeld.

4.1.1 Backus-Naur Formulering (BNF)

Deze formulering is in gebruik sinds 1960 bij de definitie van ALGOL60 en is gebaseerd op de definitie van syntactische categorieën in termen van andere syntactische categorieën (via de zogenaamde productieregels). Een syntactische categorie wordt steeds cursief (Eng.: italic) genoteerd en de notaties uit tabel 4.1 worden gebruikt.

$::=$	”te herschrijven als”
	of
$\{ \}_1$	kies 1 alternatief van de opties tussen { }
$\{ \}_{0+}$	herhaal de items tussen { } ≥ 0 keer
$\{ \}_{1+}$	herhaal de items tussen { } ≥ 1 keer
$\{ \}_{opt}$	items tussen { } zijn optioneel (0 of 1 keer)
”de rest”	”niet te herschrijven taalsymbolen”

Tabel 4.1: Overzicht BNF Syntax.

In volgend voorbeeld wordt dit geïllustreerd aan de hand van het vastleggen van de syntactische categorie *identifier*, die gebaseerd is op de syntactische categorieën *digit*, *letter* en *underscore*.

```

digit ::= 0|1|2|3|4|5|6|7|8|9
letter ::= lowercase_letter | uppercase_letter
lowercase_letter ::= a|b|c|...|z
uppercase_letter ::= A|B|C|...|Z
underscore ::= _
identifier ::= {letter|underscore}1{letter|underscore|digit}0+

```

Deze definitie legt dus formeel vast dat een identifier niet met een digit kan beginnen.

Een ander voorbeeld is de syntax van een **if** opdracht, die als volgt wordt vastgelegd:

```

if_opdracht ::= if (uitdrukking) opdracht
                  {else opdracht}opt

```

en een **while** lus wordt als volgt vastgelegd:

```

while_opdracht ::= while (uitdrukking) opdracht

```

waarbij *uitdrukking* en *opdracht* nog verder vastgelegd worden.

4.1.2 Commentaar

Commentaar in C is tekst genoteerd tussen overeenkomstige /* en */ -paren. Deze tekst wordt niet door de compiler behandeld en is zuiver een hulpmiddel voor annotatie van code.

Nesten van commentaar is niet toegelaten: beschouw volgend voorbeeld met geneste commentaar:

```

/*
  deel 1 commentaar
/*
    deel 2 commentaar
*/
  deel 3 commentaar
*/

```

De compiler zoekt de bijhorende */ bij de eerste /* en aanziet de */ net na **deel 2 commentaar** als het afsluitend teken voor de commentaar. **deel 3 commentaar** wordt dan mee gecompileerd en wanneer de laatste */ tegengekomen wordt, genereert dit een compileer-fout (afsluitend */-teken zonder bijhorend /*-teken).

Commentaar van één lijn na // (zoals in Java en C++ wel toegelaten is) is **niet** toegelaten in C.

Het is meestal niet verstandig om tijdens het debuggen code uit te commentariëren (omdat men ze later soms vergeet weer in te voegen). Een veel handiger oplossing in C is het gebruik van de conditionele compilatie opties van de preprocessor (cfr. Hoofdstuk 6).

4.1.3 Programmeerstijl: commentaar

Het is belangrijk om voldoende commentaar in code te voorzien. In een typisch C-bestand is deze meestal als volgt gestructureerd (gegevens tussen < > dienen nog ingevuld te worden):

```
*****
<naam bestand> -- <korte zin over inhoud>
Copyright (c) <datum>, <naam eigenaar of firma>.
<eventueel vermelding licentie-model>.
<eventueel iets langere beschrijving inhoud>
Release notes:
Initial implementation <datum>, <naam auteur(s)>
*****
```

```
#include <bestand1.h>      /* korte zin waarom dit bestand geincludeerd wordt*/
#include <bestand2.h>      /* idem, bijvoorbeeld welke datastructuren */
#include <bestand3.h>      /* of functies er uit elk bestand gebruikt wordt */
...
<declaratie variabelen>  /* korte zin waarom elke variabele aangemaakt wordt*/
```

```
<declaratie functie1>
*****
```

```
<beschrijving van de inhoud van de functie>
Inputs:
<uitleg over de input argumenten van de functie, void indien er geen zijn>
Outputs:
<uitleg over de output argumenten van de functie, none indien er geen is>
Returns:
<uitleg over de betekenis van de return waarde, ook wel
"command completion status" genoemd, bijv. indien het type
van de return waarde int is, korte uitleg over de betekenis
van de verschillende gehele waarden, die kunnen optreden.>
*****
```

```
<declaratie functie2>
*****
```

```
...
```

```
...
```

Bemer dat de `#include <bestand.h>` uitdrukking vergelijkbaar is met de `import` uitdrukking in Java. Een bestand-heading wordt dikwijls in combinatie met een CVS

(Concurrent Version System) gebruikt, waarbij een aantal gegevens automatisch ingevuld worden (zoals de bestandsnaam, datum, etc.). CVS-systemen komen verder aan bod in deel IV.

4.1.4 Identifiers (namen)

Identifiers zijn namen die aan variabelen of functies gegeven worden. Zoals hierboven in het BNF voorbeeld werd aangegeven, kunnen deze starten met een letter of een underscore en bevatten dan een willekeurig aantal letters, underscores of cijfers.

Volgens de ANSI C standaard worden minstens 31 karakters van een identifier behouden (langere namen die slechts na karakter 31 verschillend zijn, kunnen dus door de compiler als gelijk aanzien worden, hetgeen resulteert in een compileer-fout: *duplicate variable declaration*).

Namen die starten met een underscore worden vaak gebruikt in code van het besturingssysteem of drivers (om overlap in naamkeuze met andere code te vermijden).

Volgende stijl wordt in C bijna steeds gebruikt voor keuze van namen van variabelen en functies:

- gebruik geen hoofdletters,
- woordseparatie via " _ ",
- gebruik zinvolle namen (bijvoorbeeld geen qqr, aatp, vvr),
- naam�engte typisch tussen 5 en 15 karakters.

4.1.5 Constanten

In C kunnen ook constanten worden gedeclareerd, men maakt hierbij onderscheid tussen:

- Numerieke constanten, bijv. integere en floating point constanten.
- Karakter constanten: deze worden tussen enkele aanhalingstekens (Eng.: quotes) genoteerd, bijv. 'a', 'y',
- String constanten: voor de voorstelling van schrifttekst, genoteerd tussen dubbele quotes, bijv. "dit is schrifttekst".

Het escape karakter '\' wordt gebruikt voor speciale symbolen, zoals bijvoorbeeld in tabel 4.2.

4.2 Operatoren en precedentie

4.2.1 Operatoren en punctuatoren

Enkele voorbeelden van operatoren zijn o.a. `++`, `--`, `+`, `-`, `*`, `/`, `%`, `<`, `<=`, `>`, `=`, `? :`. Punctuatoren behelzen naast operatoren ook witte ruimte en speciale karakters: `;`, `,`, `(`, `)`, `{`, `}`, `[`, `]`. De verschillende types van operatoren, die kunnen onderscheiden worden, zijn weergegeven in tabel 4.3.

Karakter	Betekenis
'\n'	nieuwe lijn
'\t'	tabulatie
'\''	karakter '
'\\'	karakter \

Tabel 4.2: Voorbeelden van gebruik van het escape karakter '\'.

haakjes)
auto in/de-crement operatoren	++ --
rekenkundige operatoren	+ - * / %
relationele operatoren	== != < > <= >=
assignatie-operator	=
(de)referentie-operatoren	* &
logische operatoren	! &&
bitoperatoren	<< >> &
conditionele operator	?:
komma-operator	,

Tabel 4.3: Overzicht van de types operatoren.

4.2.2 Precedentieregels en associativiteit

Precedentieregels leggen vast welke operatoren voorrang hebben in volgorde van evaluatie (wanneer meerdere operatoren in dezelfde uitdrukking voorkomen). Associativiteit legt vast of de uitdrukking met de operatoren van links naar rechts of van rechts naar links uitgevoerd wordt. Tabel 4.4 geeft de precedentie (een operator in een rij heeft telkens voorrang op een operator in een lagere rij) en associativiteit weer van alle operatoren. Sommige operatoren komen pas verder in dit hoofdstuk (bijv. ++, --, &(adres) of *(dereferentie)) aan bod en anderen (bijv. ->) komen pas in een verder hoofdstuk (bijv. sectie 7.8.1) aan bod.

4.3 Variabelen

Gegevens in het geheugen worden voorgesteld door een reeks 1-en en 0-en. Elke variabele wordt gestockeerd op een bepaalde positie, ook wel het adres van die variabele genoemd. De symbolische naam van een variabele (gegeven bij declaratie) wordt geassocieerd met de geheugenlocatie. Elke variable heeft een bepaalde waarde (ook wel de *rvalue* genoemd) en wordt gestockeerd op een bepaald adres (ook wel *lvalue* genoemd).

De *rvalue* (Eng.: right of read value) is het attribuut dat de waarde van de variabele aangeeft en de *lvalue* (Eng.: left of location value) is het attribuut dat het adres van de variabele aangeeft.

Operatoren	Associativiteit
<code>() [] -> . +(postfix) --(postfix)</code>	links naar rechts
<code>++(prefix) --(prefix) !~ sizeof(type)</code>	rechts naar links
<code>+(unair) -(unair) &(adres) *(dereferentie)</code>	
<code>* / %</code>	links naar rechts
<code>+ -</code>	links naar rechts
<code><< >></code>	links naar rechts
<code>< <= > >=</code>	links naar rechts
<code>== !=</code>	links naar rechts
<code>&</code>	links naar rechts
<code>^</code>	links naar rechts
<code> </code>	links naar rechts
<code>&&</code>	links naar rechts
<code> </code>	links naar rechts
<code>? :</code>	rechts naar links
<code>= += -= *= /= >>= <<= &= ^= =</code>	rechts naar links
<code>, (komma operator)</code>	links naar rechts

Tabel 4.4: Operator precedentie en associativiteit

4.3.1 Declaratie van gegevens

De declaratie van een variabele gebeurt als volgt (in BNF syntax):

```
type identifier {= waarde}opt {, identifier {= waarde}opt}0+;
```

Bemerk dat er geen garantie is op default initialisatie!

Een andere belangrijke eigenschap van C is dat declaratie-opdrachten **steeds** voor alle andere opdrachten staan (in tegenstelling tot C++ en Java).

4.3.2 Uitdrukkingen (Eng.:expressions)

Een uitdrukking is elke zinvolle combinatie van constanten, variabelen, operatoren of functie-oproepen. Een uitdrukking heeft telkens een waarde, een type en is geen opdracht (Eng.: statement).

Een uitzondering hierop zijn uitdrukkingen van het type **void**, welke geen waarde hebben.

4.3.3 Opdrachten (Eng.:statements)

Een opdracht specificeert een uit te voeren taak. Er zijn twee types:

1. expressie-opdracht: **uitdrukking;**
2. assignatie-opdracht: **variabele=uitdrukking;**

In volgende sectie wordt verder ingegaan op assignatie-opdrachten.

4.3.4 Assignatie-opdracht

Bij een assignatie-opdracht wordt de uitdrukking in het rechterlid geëvalueerd en vervolgens wordt de berekende waarde gestockeerd in de variabele van het linkerlid. De oude waarde van de variabele in het linkerlid gaat hierbij verloren.

Beschouw bijvoorbeeld volgende assignatie:

```
a = (b=2) + (c=3);
```

als gevolg hiervan krijgen a, b en c nieuwe waarden (5, 2, 3 respectievelijk).

4.3.5 Samentrekking-operatoren

Volgende uitdrukkingen:

```
variabele += uitdrukking;
variabele -= uitdrukking;
variabele op= uitdrukking;
```

komen overeen met:

```
variabele = variabele + (uitdrukking);
variabele = variabele - (uitdrukking);
variabele = variabele op (uitdrukking);
```

waarbij op algemeen een operator voorstelt.

4.3.6 Auto-(in/de)crement operatoren

De auto-increment en auto-decrement operatoren (++ en -- respectievelijk) kunnen in zowel postfix (vermelding operator na een variabele) als prefix mode (vermelding operator voor een variabele) gebruikt worden:

- postfix: `variabele++` of `variabele--` heeft als gevolg dat de variabele pas aangepast wordt na de expressie en dus in de expressie met deze operatoren de huidige (niet-aangepaste) waarde in rekening gebracht wordt.
- prefix: `++variabele` of `--variabele` heeft als gevolg dat de variabele direct aangepast wordt en dus in een expressie de aangepaste waarde direct in rekening gebracht wordt.

Er is een belangrijke **caveat** bij het gebruik van deze auto-(in/de)crement operatoren: gebruik deze operatoren **nooit** op variabelen die (i) meer dan éénmaal voorkomen in een uitdrukking of (ii) in meer dan één argument van een functie-oproep voorkomen. Volgende voorbeelden illustreren dit:

```
a = a++ + (b=8);
c = (c++)++;
printf("%d %d",n,n*n++);
a = n/2 + 4*(1+n++);
y = n++ + n++;
```

In deze voorbeelden is het niet duidelijk (compiler- en platformafhankelijk, dus geen porteerbare code) wat de waarde van de variabelen **a** en **b** (regel 1), **c** (regel 2), **n** (regel 3), **a** en **n** (regel 4), **y** en **n** (regel 5) zal zijn.

4.4 Controle uitdrukkingen

De **if**, **else**, **do**, **while** en **switch** uitdrukkingen (Eng.: statements) zijn in C identiek aan deze van Java. Er is echter één verschil: in Java dient de conditionele expressie van het type **boolean** te zijn, in C kan dit elk type zijn (er wordt hiervoor zeer veel van het **int** type gebruik gemaakt). Volgende code is perfect mogelijk en wordt zeer veel gebruikt in C:

```
int i;

/* i krijgt een bepaalde waarde*/

if(i){
    ... /* uitgevoerd als i verschillend is van 0*/
} else {
    ... /* uitgevoerd als i gelijk is aan 0*/
}
```

Dit kan ook voor andere dan integere types, bijvoorbeeld:

```
double d;

/* d krijgt een bepaalde waarde*/

if(d){
    ... /* uitgevoerd als d verschillend is van 0.0*/
} else {
    ... /* uitgevoerd als d gelijk is aan 0.0*/
}
```

ook voor struct-types (zie verder sectie 4.6) is deze constructie mogelijk.

In tegenstelling tot Java laat C wel een komma operator toe in de test-sectie van een **for**-uitdrukking. In C kan je ook geen variabelen definiëren in de initialisatie-sectie van een **for**-lus (hetgeen bij Java wel kan).

C laat ook toe om in de code zogenaamde labels te plaatsen (zoals bij de **switch** opdracht) en via de **goto** opdracht de uitvoering van het programma naar dit label te forceren.

4.5 Fundamentele datatypes

C laat gebruik van volgende sleutelwoorden (Eng.: key words) toe om fundamentele datatypes te declareren: `char`, `int`, `float`, `double`, `signed`, `unsigned`, `long`, `short` en `void`. De verschillende toegelaten combinaties worden weergegeven in tabel 4.5, samen met hun eventuele afkorting. De betekenis van deze combinaties hangt af van de compiler en het gebruikte platform (in tegenstelling tot Java).

In C zijn alle fundamentele types numeriek, ook karakters en "logische" (booleaanse) types. Alle gehele types bestaan in twee versies met en zonder teken (`signed` en `unsigned`). Een `signed` versie van een type verschilt van de `unsigned` versie van dit type doordat het bereik verschilt.

4.5.1 Data type modifiers

Naast `signed` en `unsigned` kunnen ook volgende zogenaamde *modifiers* toegevoegd worden aan een geheel type: `short` en `long`. Bij de reële types kan de modifier `long` toegevoegd worden aan het type `double`.

Dikwijls worden verkorte notaties gebruikt indien duidelijk is welk type bedoeld wordt. Het bereik van de types hangt af van (i) de opslagruimte, die voorzien is voor dit type en (ii) het gebruik van het sleutelwoord `unsigned`. Indien er N bits ter beschikking zijn, dan is het bereik van een `unsigned` type:

$$0 \dots 2^N - 1$$

datatype voluit	datatype afgekort
<code>char</code>	<code>char</code>
<code>signed char</code>	<code>signed char</code>
<code>unsigned char</code>	<code>unsigned char</code>
<code>short int</code>	<code>short</code>
<code>signed short int</code>	<code>short</code>
<code>unsigned short int</code>	<code>unsigned short</code>
<code>int</code>	<code>int</code>
<code>signed int</code>	<code>int</code>
<code>unsigned int</code>	<code>unsigned</code>
<code>long int</code>	<code>long</code>
<code>signed long int</code>	<code>long</code>
<code>unsigned long int</code>	<code>unsigned long</code>
<code>float</code>	<code>float</code>
<code>double</code>	<code>double</code>
<code>long double</code>	<code>long double</code>

Tabel 4.5: Lijst van fundamentele datatypes in C en hun afkortingen.

en het bereik voor een `signed` type:

$$-2^{N-1} \dots 2^{N-1} - 1$$

Gegevens in een `signed` type worden in 2-complement voorstelling opgeslagen. C garandeert **steeds** dat het `int`-type één machinewoord beslaat (dus 4 bytes op een 32-bits machine en 8 bytes op een 64-bits machine). Verder wordt er **steeds** gegarandeerd dat de lengte van het `short`-type kleiner of gelijk is aan de lengte van het `int`-type en de lengte van het `long`-type groter of gelijk is aan de lengte van het `int`-type. Overflow in C wordt niet gedetecteerd noch gemeld! Vermits de lengte van type (en dus het bereik) machine-afhankelijk is, is er een operator `sizeof` voorzien, die op een type kan toegepast worden om de *lengte in bytes* van dit type op te vragen. Volgende code:

```
int a,b,c,d,e;
a=sizeof(short);
b=sizeof(int);
c=sizeof(long);
d=sizeof(double);
e=sizeof(long double);
```

zorgt ervoor dat de variabelen `a`, `b`, `c`, `d`, en `e` als respectievelijke waarden krijgen 2, 4, 8, 8 en 16 (op een typisch 32-bit platform, maar het resultaat is uiteraard platformafhankelijk).

4.5.2 Constante gehele en reële types

Een gehele constante wordt als volgt voorgesteld (BNF syntax):

$\{\text{prefix}\}_{\text{opt}} \{+|- \}_{\text{opt}} \text{getal} \{\text{suffix}\}_{\text{opt}}$

De prefix kan hierbij de waarden aannemen uit tabel 4.6 en de suffix kan de waarden aannemen uit tabel 4.7.

Een reële constante wordt als volgt genoteerd (BNF syntax):

$\{+|\-\}_{\text{opt}} \text{getal.getal} \{\{\text{e|E}\} \{+|\-\}_{\text{opt}} \text{getal}\}_{\text{opt}} \{\text{suffix}\}_{\text{opt}}$

De suffix kan hierbij de waarden aannemen uit tabel 4.8. Bemerk dat bij reële types zowel *underflow* als *overflow* mogelijk zijn!

4.5.3 Typeconversie

Wanneer twee of meer verschillende types in een uitdrukking gebruikt worden, wordt er een typeconversie uitgevoerd. Hierbij worden volgende regels toegepast:

<i>geen prefix</i>	decimale constante
0	octale constante
0x	hexadecimale constante

Tabel 4.6: Betekenis van prefix bij een gehele constante.

<i>geen suffix</i>	int
u U	unsigned
l L	long
ul UL	unsigned long

Tabel 4.7: Betekenis van suffix bij een gehele constante.

<i>geen suffix</i>	double
f F	float
l L	long double

Tabel 4.8: Betekenis van suffix bij een reële constante.

- wanneer er een **unsigned char**, een **signed char** of een **short** in de uitdrukking voorkomt wordt deze geconverteerd naar een **int**, indien het bereik dit toelaat, zoniet wordt er geconverteerd naar een **unsigned int**,
- onderstaande tabel 4.9 wordt gebruikt voor conversie van de andere types: er wordt verondersteld dat een binaire operator (met een linker- en rechterlid) gebruikt wordt, in de tabel voorgesteld door **+**. De tabel wordt van boven naar onder overlopen tot wanneer één van beide types (i.e. linkerlid of rechterlid) tegengekomen wordt (**x** stelt hierbij het andere type van gelijke of lagere volgorde voor).

voor conversie	na conversie
long double + x	long double + long double
double + x	double + double
float + x	float + float
unsigned long + x	unsigned long + unsigned long
long + unsigned	long + long unsigned long + unsigned long
long + x	long + long
unsigned + x	unsigned + unsigned
x + x	int + int

Tabel 4.9: Impliciete type conversies (**x** = het andere type van gelijke of lagere volgorde).

De conversie van een **long** gecombineerd met een **unsigned** hangt af van het feit of het bereik van **long** voldoende is om de gegevens voor te stellen, zoniet wordt naar **unsigned long** geconverteerd.

4.5.4 Cast

Indien men explicet een type conversie wil forceren, kan men gebruikmaken van een cast. De syntax hiervoor is:

```
(type) uitdrukking
```

Een voorbeeld hiervan is volgende code:

```
double d1 = 39.1;
int i = (int)d1;
double d2 = (double)(i=97);
```

Het wordt sterk aanbevolen om expliciete casts te gebruiken: het geeft namelijk duidelijk aan wat de bedoeling is (zeer handig indien andere ontwikkelaars de code bekijken en wanneer men na een tijdje eigen code nog eens opnieuw bekijkt). Impliciete conversieregels zijn in veel gevallen systeemafhankelijk en kunnen dus tot verwarring leiden.

4.6 Struct's ipv klasses

4.6.1 struct declaratie

Een **struct** is de groepering van gegevens van verschillende types (HETEROGENE types) die logisch samenhangen (bijv. ze betreffen dezelfde fysieke entiteit) in een nieuw datatype. Volgende syntax wordt gebruikt om een **struct** te declareren:

```
struct naam {
    type_1 naam_1;
    type_2 naam_2;
    ...
};
```

`naam_1, naam_2, ...` zijn velden van de struct `naam`. Als velden zijn ook wijzers (Eng.: pointers), rijen (Eng.: arrays) of structs (nesting) toegelaten.

De struct declaratie legt de blauwdruk vast van toekomstige variabelen van type **struct naam**. Aanmaken van variabelen van dit type gebeurt dan als volgt:

```
struct naam var1, var2;
```

Een alternatieve manier om een variabele van een struct type aan te maken is door combinatie met de struct-definitie:

```
struct naam {
    type_1 naam_1;
    type_2 naam_2;
    ...
} var1, var2;
```

hierdoor worden onmiddellijk twee variabelen van het nieuwe type aangemaakt.

Vermits C niet object-georiënteerd is, kan men geen klassen declareren en gebruiken: een **struct** laat toe om data te groeperen, maar laat **niet** toe dat functies in de declaratie opgenomen worden (zoals methoden in een klasse-declaratie).

4.6.2 C-idioom

In C worden struct-declaraties heel vaak voorafgegaan van het sleutelwoord **typedef** om op deze manier een nieuwe type te declareren. Vermits dit typisch is voor C, wordt dit een C-idioom genoemd. Beschouw als voorbeeld volgende declaratie:

```
typedef struct {
    type_1 naam_1;
    type_2 naam_2;
    ...
} naam;
```

dan kunnen variabelen van dit type op de volgende manier aangemaakt worden:

```
naam var1, var2;
```

met andere woorden: het sleutelwoord **struct** kan in de declaratie van de variabelen weggelaten worden, hetgeen natuurlijker omgaan met struct-types toelaat.

4.6.3 Toegang tot gegevens

De operator **.** laat toe om de individuele velden van een struct te benoemen. De variabele **naam_struct_variabele.naam_veld** mag overal gebruikt worden waar een uitdrukking van hetzelfde type toegelaten is. Volgende code illustreert het aanmaken van een variabele van het type **persoon** en het invullen van twee velden:

```
typedef struct{
    char naam[30];
    short leeftijd;
    char code;
} persoon;

int main(void){
    persoon p1;
    p1.leeftijd=30;
    p1.code='A';
    printf("Geef nieuwe code in voor p1 :");
    scanf("%c",&(p1.code));
}
```

De functie **scanf** komt aan bod in sectie 4.8.2.

4.6.4 operator =

De assignatie-operator (=) toegepast op struct-variabelen, zorgt voor het kopiëren van alle velden van het rechterlid naar het linkerlid. Volgende code illustreert het gebruik van deze operator:

```
#include <stdio.h>
#include <string.h>
typedef struct{
    char naam[30];
    short leeftijd;
    char code;
} persoon;

void print_persoon(persoon);

int main(void) {
    persoon p1,p2;
    p1.leeftijd=30;
    p1.code='A';
    strcpy(p1.naam,"Jan Janssen");
    print_persoon(p1);
    p2=p1;
    print_persoon(p2);
    p2.naam[1]='o';
    print_persoon(p1);
    print_persoon(p2);
}

void print_persoon(persoon p) {
    printf("Naam : %s\nLeeftijd : %d\nCode : %c\n",
           p.naam,p.leeftijd,p.code);
}
```

4.6.5 Initialisatie van een struct

De syntax voor het initialiseren van een struct is sterk gelijkaardig aan de syntax voor het initialiseren van een rij (Eng.: array). De initialisatie-uitdrukking voor elk veld wordt genoteerd tussen {}. Indien expliciete initialisatie gebruikt wordt (aan de hand van de {} bij declaratie van de variabelen), dan worden alle niet gespecificeerde velden automatisch op 0 (alle bits op 0) geplaatst. In volgende code wordt dit geïllustreerd:

```
persoon p={"Jan Janssen",30,'A'};
persoon q={"Piet Janssen"};
persoon r[5]={{"Mario Puzo",55,'P'}, {"Don Corleone"}};
persoon n[10]={0}; /* alles 0 ! */
```

In regel 1 worden alle velden geïnitialiseerd. In regel 2 wordt enkel het eerste veld een waarde toegekend, en worden de twee andere velden automatisch op 0 geplaatst. In

regel 3 wordt een rij van 5 struct-variabelen aangemaakt, waarvan het eerste element uit de rij volledig geïnitialiseerd wordt met de opgegeven waarden, van het tweede element wordt enkel het eerste veld een gespecificeerde waarde toegekend en dus worden de andere twee velden automatisch op 0 geplaatst. Van de drie andere elementen uit de rij worden ook alle velden op 0 geplaatst.

In regel 4 wordt een array van 10 struct-variabelen aangemaakt, waarvan **alle** velden op 0 geplaatst worden.

4.6.6 Struct's: samenvatting

C is een niet OO (Eng.: Object Oriented) taal, in tegenstelling tot Java. Het is dus logisch dat er geen **class** types kunnen gedeclareerd worden. Voor de groepering van attributen kan je in C wel structuren definiëren aan de hand van het sleutelwoord **struct**. Een C **struct** kan je best vergelijken met een Java **klasse** die enkel publieke data attributen bevat en geen methoden. Java zelf kent geen **struct**'s.

Members van een C **struct** kunnen aangesproken worden via de **.** operator (net zoals in Java). **struct**'s kunnen ook geïnitialiseerd worden bij hun declaratie. Toekennen van twee **struct**'s aan elkaar zorgt voor het kopiëren van de members van de ene **struct** naar de andere (in tegenstelling tot Java, waar de assignatie de referenties aan elkaar gelijk stelt).

4.7 Union's

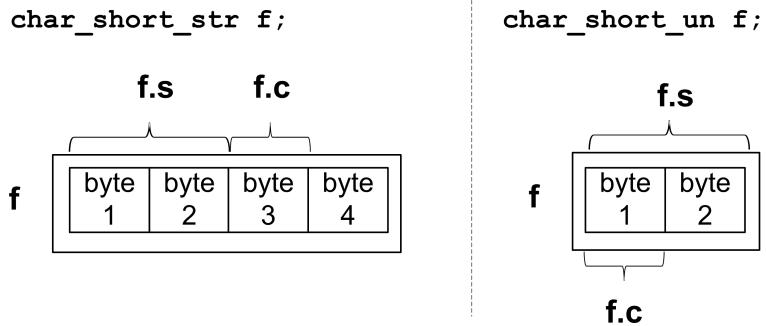
Een union wordt gebruikt in plaats van een struct wanneer **slechts één** van de velden op elk ogenblik gebruikt wordt. Het vereiste geheugen voor een union is dus gelijk aan de lengte van het grootste veld (in bytes uitgedrukt) en afgerond naar de woordlengte van de processor. Bij een struct daarentegen is het vereiste geheugen gelijk aan de som van de lengte van alle velden (eveneens in bytes uitgedrukt en het totaal afgerond naar de woordlengte van de processor).

Beschouw volgend voorbeeld van een struct- en union declaratie:

```
typedef struct char_short_str {
    short s;
    char c;
} char_short_str;

typedef union char_short_un {
    short s;
    char c;
} char_short_un;
```

In dit voorbeeld neemt een variabele van het type **char_short_un** dus minder plaats in dan een variabele van het type **char_short_str**, maar kan slechts één van de twee velden (**s** of **c**) gebruikt worden. Figuur 4.1 geeft schematisch het ingenomen geheugen voor beide gevallen weer.



Figuur 4.1: Voorstelling in het geheugen van een variabele van het type `char_short_str` en `char_short_un`.

Volgende code toont een uitgebreider voorbeeld waarbij struct's en een union gebruikt worden voor het bijhouden van bibliotheek-items (in dit voorbeeld boeken of CD's). De struct `bib_item` bevat een union en een bijhorende `int`-variabele, die aangeeft of het `bib_item` een boek of een CD bevat (aan de hand van de conventie: 0 duidt erop dat de union een boek bevat en 1 duidt op het bevatten van een CD).

```

#define N 30

typedef struct boek {
    char titel[N];
    int aantal_blz;
    int prijs;
} boek;

typedef struct cd {
    char titel[N];
    int aantal_tracks;
    int speeltijd;
    int prijs;
} cd;

typedef union boek_cd {
    boek b;
    cd c;
} boek_cd;

typedef struct bib_item {
    int b_cd /* 0 voor boek 1 voor cd */
    boek_cd item;
} bib_item;

```

In volgende code wordt een rij van 20 `bib_item`'s aangemaakt, die afwisselend een boek en CD bevatten (via de `%` of *rest bij deling door*-operator). Vervolgens wordt de prijs van een CD op 10 geplaatst en de prijs van een boek op 20.

```

int main(void) {
    int i;
    bib_item item[20];
    for(i=0;i<20;i++) {
        item[i].b_cd=i%2;
        /* even -> boek */
        if(item[i].b_cd)
            item[i].item.c.prijs=10;
        else item[i].item.b.prijs=20;
    }
}

```

Bemerkt dat voor het modeleren van dergelijke gegevens beter de principes van overerving en polymorfisme gebruikt worden. Deze zijn echter enkel mogelijk in object-georiënteerde talen, en dus niet in C. C++ is wel object-georiënteerd: in het deel C++ komen we uitgebreid op overerving en polymorfisme terug. In C is het gebruik van unions aangewezen voor het modeleren van gegevens die verschillende vormen kunnen aannemen (zoals een `bib_item` dat zowel een `boek` als een `cd` kan voorstellen).

Net zoals `struct`'s kent Java ook geen `union`'s.

4.8 Elementaire I/O

4.8.1 uitvoer: `printf` functie

De `printf` functie wordt in C gebruikt voor output van gegevens naar het scherm (i.e. afdrukken van gegevens). Om gegevens naar een bestand te schrijven wordt de `fprintf` functie gebruikt. Beiden zijn gedeclareerd in het headerbestand `stdio.h` en vereisen dus `#include <stdio.h>` bovenaan het bestand.

De syntax van `printf` is als volgt:

```
printf(formaat_string, argument1, argument2,...)
```

Voor elk argument wordt het formaat in `formaat_string` gespecificeerd door een conversiekarakter telkens na %. De mogelijke conversiekarakters zijn weergegeven in tabel 4.10 ([1] betekent dat 1 optioneel vermeld kan worden). De argumenten worden één voor één in de `formaat_string` gesubstitueerd.

4.8.2 invoer: `scanf` functie

De `scanf` functie wordt in C gebruikt voor input van gegevens via het toetsenbord. Om gegevens uit een bestand te lezen wordt de `fscanf` functie gebruikt. Beiden zijn ook gedeclareerd in het headerbestand `stdio.h` en vereisen dus ook `#include <stdio.h>` bovenaan het bestand.

De syntax van `scanf` is als volgt:

c	karakter
[1]d	(long) decimaal geheel getal
[1]o	(long) octaal geheel getal
[1]x	(long) hexadecimaal geheel getal
e	reëel getal in wetenschappelijke notatie
f	reëel getal
g	kortste van e- of f-formaat
s	string
%	teken '%'

Tabel 4.10: Conversie-karakters voor de `printf` en `scanf`-functie.

```
scanf(formaat_string, argument1, argument2,...)
```

Voor elk argument wordt het formaat in `formaat_string` gespecificeerd door een conversiekarakter telkens na % (identiek zelfde conversiekarakters als bij `printf`, weergegeven in tabel 4.10). De argumenten stellen steeds adressen voor (&-operator) (vermits de argumenten door de functie-oproep aangepast worden, cfr. call-by-value en call-by-reference principes, die in hoofdstukken 5 en 7 aan bod komen).

`scanf` negeert witte ruimte, behalve bij lezen van tekst. De return-waarde van `scanf` is een geheel getal (type `int`), die het aantal succesvol gelezen variabelen bevat.

Volgende code illustreert het inlezen van een geheel getal en het opslaan van het ingelezen getal in de variabele i.

```
int i=0;
printf("Geef een getal :");
scanf("%d",&i);
```

In volgend voorbeeld houdt de variabele b het aantal succesvol ingelezen variabelen bij en wordt de waarde van b ook afgedrukt.

```
#include <stdio.h>

int main(void)
{
    int i=1,j=2;
    int b;
    b=scanf("%d %d",&i,&j);
    printf("res=%d\n",b);
    return 0;
}
```

4.8.3 Macro's `getchar` en `putchar`

Deze beide macro's zijn gedeclareerd in `<stdio.h>` en dienen voor het inlezen (`getchar`) en uitschrijven (`putchar`) van één karakter (eenvoudiger syntax dan `scanf` en `printf`).

`getchar` geeft EOF (Eng.: End Of File) als het einde van de input bereikt is (komt overeen met Ctrl-Z).

Macro's komen in sectie 5.9 uitgebreid aan bod.

4.9 Wiskundige functies

Deze zijn gedeclareerd in het headerbestand `math.h` en vereisen dus `#include <math.h>`. Volgende functies kunnen gebruikt worden:

```
double sqrt(double)
double pow(double,double)
double exp(double)
double log(double)
double sin(double)
double cos(double)
double tan(double)
```

`sqrt` neemt de vierkantwortel van het argument, `pow` geeft argument1 tot de macht argument2 terug, `exp` berekent e (2.71828...) tot de macht het argument en `log` geeft het logaritme met grondtal e van het argument terug.

Hoofdstuk 5

C Functies en Macro's

In dit hoofdstuk komt de manier aan bod om modulariteit in C programma's in te bouwen: aan de hand van functies en macro's. Het principe van call-by-value wordt toegelicht (call-by-reference komt verder in hoofdstuk 7 aan bod), en de betekenis van de *storage classes* komen aan bod. Verder wordt ook behandeld hoe men een functie met een variabel aantal argumenten kan schrijven (i.e. waarbij men niet op voorhand het aantal argumenten vastlegt).

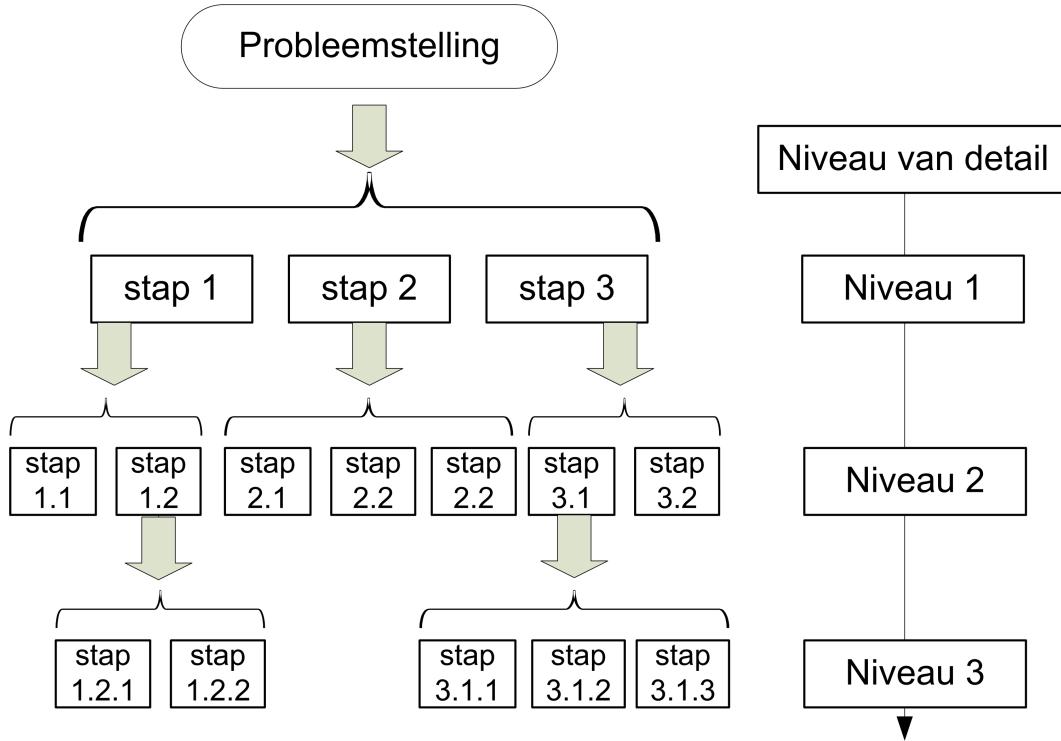
5.1 Concept functies in C

Een functie in C is het basisblok van het gestructureerd paradigma: programma's worden gestructureerd door de code op te splitsen in functies, die waar nodig opgeroepen worden. Dit wordt het principe van *Call & Return* genoemd: een functie wordt opgeroepen (Eng.: function call) en na beëindiging (Eng.: return) worden de volgende instructies uit de oproepende code uitgevoerd.

Als programmeermethodiek wordt heel vaak *stapsgewijze verfijning* gebruikt: een probleem wordt opgesplitst in verschillende onderdelen, en elk van deze onderdelen wordt verder ook opgesplitst, etc. tot wanneer men basisbouwbladen bekomt. Op deze manier bekomt men een hiërarchische structuur: op elk niveau worden functies voorzien die op hun beurt functies op een onderliggend niveau oproepen. Figuur 5.1 geeft de methodiek van *stapsgewijze verfijning* schematisch weer.

Het gebruik van functies is ook gebaseerd op het *divide et impera* principe: de complexiteit wordt beheerst door dekompositie van de code in functionele blokken (voorgesteld door functies). Ieder blok verbergt complexiteit (i.e. implementatie-details) en biedt een goed gedefinieerde interface aan voor de gebruikers (i.e. de ontwikkelaars die deze functie willen oproepen). Een gebruiker dient enkel de interface (= uitwendige black-box beschrijving) te kennen en geen implementatiедetails. Dit principe wordt soms ook *procedurale abstractie* genoemd (de termen *procedure* en *functie* worden vaak door elkaar gebruikt, abstractie betekent dat men geen onderliggende details hoeft te kennen, enkel de interface en de betekenis van functies).

Programmeren in C bestaat er dan uit om de functionele blokken samen te stellen tot



Figuur 5.1: Principe van stapsgewijze verfijning voor oplossen van (complexe) problemen in een procedurale taal zoals C.

andere met een complexere functionaliteit (door de bestaande blokken telkens op te roepen).

5.2 Programma modulariteit

5.2.1 Concept modules

Java programma's zijn meestal gebouwd op een modulaire manier, die code-hergebruik ondersteunt. De broncode is verspreid over verschillende bronbestanden met extensie `.java`, die gecompileerd worden naar Java byte-code in class-bestanden (met extensie `.class`). In Java is er een direct verband tussen de naam van een klasse en het bestand dat de code voor die klasse bevat. Deze worden tijdens de uitvoering (Eng.: at run-time) gecombineerd om het uitvoerbaar programma te vormen.

C programma's bestaan uit een verzameling functies en variabelen, die elk een bepaald type hebben. De C compiler leest elk bronbestand sequentieel. Een declaratie van een type, functie of variabele vertelt de compiler dat de naam bestaat en hoe die kan gebruikt worden later in het bestand. Indien de compiler een naam tegenkomt die geen declaratie heeft vroeger in het bestand, genereert de compiler een foutbericht of een

verwittiging omdat hij geen zicht heeft op hoe de naam gebruikt kan worden. Dit in tegenstelling tot Java, waar de compiler ook voorwaarts naar declaraties kan zoeken en zelfs in andere bronbestanden.

5.2.2 Werking C compiler

Een uitgebreid C programma wordt steeds gesplitst in verschillende bronbestanden (elk met de .c extensie), en compilatie van elk van deze levert een *object bestand* op (één per bronbestand, met de extensie .o of .obj). Dit zijn dan de modules die kunnen gecombineerd worden tot een uitvoerbaar programma. Een object-bestand bevat de namen van de functies en namen van de globale variabelen, die gedefinieerd zijn in het bijhorende bronbestand. Een object-bestand laat toe om te verwijzen naar andere functies en variabelen via hun naam, zelfs in een aparte module. In C is er geen verband tussen de namen van de functies en variabelen en de namen van de modules die hen bevatten.

5.2.3 Linker

Een uitvoerbaar C programma wordt gemaakt door alle relevante modules (als object bestanden) aan te bieden aan een *linker* (die in de meeste gevallen onderdeel is van de compilatie-omgeving). Deze linker probeert al de naamverwijzingen in de object bestanden op te zoeken om deze te kunnen associëren met de juiste functies en variabelen. Dit opzoeken zal niet lukken als sommige namen niet gevonden worden of als dezelfde naam meer dan éénmaal optreedt. Indien alle naamverwijzingen gevonden worden, zal het linken lukken en wordt er een uitvoerbaar bestand aangemaakt door de linker.

5.3 Definitie van functies

Definitie van een functie betekent het voorzien tussen { } van de nodige code, die bij oproepen van de functie uitgevoerd wordt. De syntax voor een dergelijke definitie is:

```
{type}opt functie_naam(parameter_lijst) {
    {declaratie}0+
    {opdracht}0+
}
```

Een functie-definitie bevat een formele parameterlijst, dit is een lijst van nog niet ingevulde variabelen, die bij functie-oproep ingevuld worden met de doorgegeven argumenten. Indien de types van de doorgeven argumenten en deze uit parameter_lijst verschillen worden impliciete typeconversies uitgevoerd (zoals in vorig hoofdstuk besproken). De uitvoering van functie stopt wanneer een **return** opdracht tegengekomen wordt. De syntax hiervoor is:

```
return {uitdrukking}opt;
```

De uitdrukking dient van hetzelfde type te zijn als het type van de functie (indien nodig wordt ook een typeconversie uitgevoerd).

In Java wordt een methode zonder argumenten aangeduid door `methode_naam()`. In C dient een dergelijke functie aangeduid te worden door `functie_naam (void)`. De invokatie van de functie gebeurt door `functie_naam();` De declaratie `functie_naam()` is toegelaten, maar betekent *ongespecifieerde argumenten* in plaats van *geen argumenten*. De compiler voert dan geen type-checking van de argumenten uit wanneer de functie geïnvokeerd wordt en dit wordt zeker niet aangeraden. C biedt de mogelijkheid tot definitie van functies met een variabel aantal argumenten. Een belangrijk voorbeeld hiervan is de `printf` functie. In Java is dit ook mogelijk, maar dan dienen de argumenten van hetzelfde type te zijn.

Bij `void` functies is er geen return waarde (vandaar de *opt* in de syntaxbeschrijving). Er is in een functiedefinitie uiteraard meer dan één `return` uitdrukking toegelaten (de eerste die tegengekomen wordt zorgt voor beëindiging van de functie). Indien er geen `return` uitdrukking voorzien is, stopt de uitvoering van de functie automatisch als het einde van de functie bereikt wordt.

5.4 Declaratie van functies

Een functie-declaratie ziet er uit als een functie-definitie, maar bevat geen code (de code tussen {}, inclusief de haakjes, wordt dan vervangen door een punt-komma).

Een globale variabele wordt gedefinieerd buiten een functie, terwijl een lokale variabele binnen een functie gedefinieerd wordt. De lokale variabele is enkel geldig binnen de haakjes {}, waarin ze gedefinieerd is. In tegenstelling tot Java, dienen de variabelen in het begin van een functieblok gedefinieerd te worden vooraleer ze gebruikt kunnen worden in uitdrukkingen. Een ander belangrijk verschil met Java is dus dat een variabele in een `for`-lus niet kan gedefinieerd worden binnen de initialisatie-uitdrukking van een `for`-lus.

Elke functie moet gedeclareerd worden VOOR gebruik (zoniet kan de compiler niet controleren of het type en aantal van de argumenten correct is). Hiervoor zijn er twee oplossingen:

1. definieer alle functies VOOR ze gebruikt worden (niet altijd mogelijk: wanneer bijvoorbeeld twee functies elkaar oproepen, functie `f1` roept `f2` op en omgekeerd).
2. gebruik een functie-prototype en plaats dit voor het gebruik van de functie zelf.

De syntax van een functie-prototype is:

`{type}opt functie_naam(proto_lijst);`

```
niet_lege_proto_lijst ::= {type {naam}opt} {, type {naam}opt}0+
lege_proto_lijst ::=
```

Wanneer een groot C programma gesplitst wordt over verschillende modules, zal code in één module meestal verwijzen naar een functienaam of variabelenaam in een andere module of zal de code types gebruiken, die ook in een andere module gebruikt worden. De gebruikelijke manier om dit op te lossen is bij het gebruik van elke verwijzing ook de declaratie te vermelden, die uitlegt wat de naam betekent.

Telkens zulke declaraties herhalen in elk bronbestand die ze nodig heeft, zou een zeer lastige en fouten-introducerende aanpak zijn (zeker als declaraties tijdens de ontwikkeling van een programma af en toe aangepast worden). Daarom worden deze declaraties in een apart bestand geplaatst, het headerbestand (Eng.: header file) genoemd. Een headerbestand heeft meestal de .h extensie. Een headerbestand wordt automatisch in de code ingevoegd door de preprocessor wanneer deze een **#include** commando in de broncode aantreft.

Bij programmeren in C worden functie-prototypes heel vaak in een bijhorend headerbestand geplaatst: in het .c bestand staan de definities van de functies en in het .h bestand de declaraties.

Beschouw volgend voorbeeld met declaraties van functies in het bestand *functies.h*:

```
int f1(double);
double f2(int,int);
f3(double,double);
f4(void);
```

en de definities van deze functies in het bestand *functies.c*:

```
#include "functies.h"
...
int f1(double a) {...}
double f2(int a,int b) {...}
f3(double x,double y) {...}
f4() {...}
```

en vervolgens het gebruik van deze functies in een ander bestand, waarbij door **#include "functies.h"** de functie-declaraties geïncludeerd worden:

```
#include "functies.h"
...
int g(double a, double b, int i){
    ...
    if (f1(a)>0){
        f2(i,i);
        f3(a,b);
    }
    ...
}
```

5.4.1 Traditionele C-syntax

De zogenaamde traditionele C-syntax was in gebruik voor de gestandardiseerde vorm (ANSI C). Kenmerken van deze traditionale syntax zijn: een prototypelijst zonder types en een alternatieve vorm van parameterlijst. Dit wordt geïllustreerd in volgend voorbeeld, waarbij eerst de ANSI C syntax getoond wordt voor twee functie-definities **f** en **g**:

```
int f(int a,int b) {
    ...
}

int g(double a,int b,char c) {
    ...
}
```

In traditionele C syntax komt dit overeen met (equivalent met):

```
int f(a, b)
    int a,b;
{...}

int g(a, b, c)
    int b;
    double a;
    char c;
{...}
```

Bemerk dat de parameters dus niet in volgorde hoeven opgesomd te worden.

5.4.2 Uitgebreider voorbeeld

Beschouw volgend code-voorbeeld, met 3 functie definities, waarvan de declaraties bovenaan zijn opgenomen en de functies in de main-functie opgeroepen worden.

```
#include <stdio.h>

void print_lijn(int);
void print_titel(void);
void print_ascii_tabel(char,char);

int main(void) {
    print_lijn(40);
    print_titel();
    print_ascii_tabel('A','z');
    print_lijn(40);
    return 0;
}

void print_lijn(int n){
```

```

int i=0;
putchar('\n');
for(i=0;i<n;i++)
    putchar('-');
putchar('\n');
}

void print_titel(void) {
    printf("\nDe ASCII-tabel : \n");
}

void print_ascii_tabel(char s,char e) {
    char c;
    for(c=s;c<=e;c++)
        printf("%c : %d\t",c,c);
}

```

5.5 Oproepen van functies: call by value

Een functie-oproep in code kan overal gebruikt worden waar een uitdrukking van hetzelfde type toegelaten is. Het effect van een functie-oproep is als volgt:

1. de waarden van de argumenten worden geëvalueerd
2. de waarde van elk argument wordt toegekend aan de formele parameter van de functie-definitie
3. de controle wordt overgedragen aan functie (m.a.w. de uitvoering van de functie wordt gestart)
4. de resultaatwaarde (Eng.: return value) vervangt de functie-oproep (eventueel na type-conversie) in de oproepende code

Indien een functie-prototype gebruikt wordt, dwingt de compiler de consistentie van de formele en de actuele (i.e. doorgegeven) parameterlijst af!

Belangrijk: de waarde van een variabele wordt telkens als argument doorgegeven. Indien het argument geen adres is, kan een functie-oproep **nooit** de waarde van een variabele uit de oproepende code wijzigen. M.a.w. er wordt steeds een kopie van een variabele meegegeven bij een functie-oproep, aanpassing van een kopie verandert nooit het origineel, tenzij het om een adres gaat, waardoor de functie wel via het adres aan de variabele kan.

Volgende code illustreert het doorgeven van een variabele aan een functie:

```

#include <stdio.h>

void bepaal_opvolger(char);

int main(void) {

```

```

char a='a';
printf("a=%c\n",a);
bepaal_opvolger(a);
printf("a=%c\n",a);
return 0;
}

void bepaal_opvolger(char a) {
    a++;
}

```

Na de functie-oproep in dit voorbeeld blijft de waarde van de variabele `a` ongewijzigd.

5.6 Storage classes en scope

Een variabele wordt steeds gekenmerkt door zijn:

1. type (opgegeven bij declaratie)
2. zichtbaarheid
3. levensduur

De laatste twee worden bepaald door:

1. de plaats van declaratie (bereik-regels of Eng.: scoping rules)
2. de opslagmodaliteiten (opslagklasse of Eng.: storage class)

De basis bereik-regel is dat een variabele zichtbaar is in het blok, waarin de variabele gedeclareerd is. Een blok is gedefinieerd als een code blok tussen accolades (`{}`). Er zijn vier sleutelwoorden (Eng.: key words) die de opslagklasse vastleggen (deze komen verder in deze sectie aan bod). Door keuze van een opslagklasse wordt mogelijks afgeweken van de basis bereik-regel.

5.6.1 Name hiding/masking

Als een naam van een variabele in een genest blok identiek is aan de naam van een variabele in een buitenblok, dan is de variabele in het buitenblok gemaskeerd en wordt in het binnenvlokk met de dichtstbijzijnd gedeclareerd variabele gewerkt. Beschouw volgend voorbeeld, waarbij de variabelen `a` en `b` ook gedeclareerd worden in het binnenvlokk:

```

{...
int a=1;
int b=2;
{
    int a=10;
    float b=20.0;
    printf("a=%d b=%f",a,b);
}

```

```

    }
    printf("a=%d b=%d",a,b);
}

```

In het binnenste blok worden in dit voorbeeld de waarden 10 en 20.0 afgedrukt en in het buitenste blok de waarden 1 en 2.

Het gebruik van extra blokken (code tussen extra {}) kan zorgen voor geheugenbesparing (bij beëindiging van het blok kunnen de in dit blok gedeclareerde variabelen niet meer gebruikt worden en kunnen ze direct opgeruimd worden, i.e. het ingenomen geheugen kan vrijgegeven worden). Geneste blokken worden ook vaak gebruikt bij het debuggen van programma's (men kan nieuwe variabelen in dit blok declareren zonder gevaar te lopen dat een variabele met dezelfde naam reeds bestaat).

5.6.2 Storage classes

Volgende vier opslagklassen kunnen onderscheiden worden (het overeenkomstige sleutelwoord wordt steeds voor de declaratie van de variabele geplaatst):

auto

Dit is de default storage class voor variabelen binnen een blok gedeclareerd. Lokale variabelen van een blok worden gecreëerd bij de start van het blok en worden vernietigd bij het eindigen van het blok. De levensduur van de variabele is dus de levensduur van het blok. Vermits **auto** de default waarde is, kan de vermelding van **auto** steeds in declaraties weggelaten worden.

register

Een variable in deze opslagklasse declareren is een tip aan de compiler om de variabele in een processorregister (snelste vorm van geheugen) op te slaan. Dit wordt veel gebruikt voor tellers in lussen, omdat deze variabele typisch vaak gebruikt wordt en het dus de uitvoeringssnelheid ten goede komt als deze variabele snel kan uitgelezen of aangepast worden. Sommige compilers proberen variabelen van het type **int** zo veel mogelijk in een processorregister op te slaan.

Het explicet vermelden van het sleutelwoord **register** is echter geen garantie: sommige compilers negeren dit en het hangt soms af van de compiler-opties.

extern

Dit is de default storage class voor variabelen buiten een blok gedeclareerd (deze kunnen niet **auto** of **register** of zijn!). Het betreft dan een globale variabele voor het programma: de naam kan gedefinieerd zijn in dezelfde compilatie-eenheid, of in een andere compilatie-eenheid. Vermits een compiler steeds bestand per bestand compileert, is een compilatie-eenheid equivalent met een bestand.

De levensduur van **extern** variabelen is dus permanent tijdens de uitvoeringstijd van het programma. Beschouw volgend bestand met globale declaratie van variabele **a**, *bestand1.c*:

```
int a=1;

int f3(void) {
    return a=3;
}
```

In *bestand2.c* kan dezelfde variabele opnieuw gebruikt worden door gebruik van het **extern** sleutelwoord.

```
int f2(void) {
    extern int a;
    return a=2;
}
```

De linker zorgt ervoor dat beide stukken code naar dezelfde fysische variabele verwijzen. Alle functies zijn steeds **extern**. Het doorgeven van informatie aan functies gebeurt via argumenten, en niet via globale variabelen (sterk afgeraden).

static

Indien dit toegepast wordt op lokale variabelen, dan zorgt dit ervoor dat de waarde behouden wordt bij het herstarten van een blok. Het geheugen wordt na het uitvoeren van een blok dus niet vrijgegeven.

static kan ook op functies toegepast worden, vermits functies steeds **extern** zijn valt dit onder de categorie hieronder (**static extern**).

static extern

Indien dit toegepast wordt op globale variabelen, dan betekent dit dat het bereik beperkt wordt tot dezelfde compilatie-eenheid vanaf het punt van declaratie.

Indien dit toegepast wordt op een functie, dan is het gebruik van deze functie beperkt tot dezelfde compilatie-eenheid (m.a.w. een **static extern** functie kan niet buiten het bestand, waarin ze gedeclareerd is, gebruikt worden).

5.7 Enkelvoudige naamruimte voor functies en globale variabelen

Elke klasse in Java definieert een naamruimte (Eng.: namespace), die toelaat dat methoden en attributen in verschillende ongerelateerde klassen een zelfde naam hebben. In C zijn alle functies globaal en is er slechts één naamruimte. In grote projecten moet dus bijzondere aandacht besteed worden aan het uniek kiezen van de namen van functies en globale variabelen.

Let wel: een **struct**, **union** of **enum** (deze laatste komt in hoofdstuk 8 aan bod) definieert een aparte naamruimte voor zijn members en elk blok (code tussen { }) definieert een aparte naamruimte voor zijn lokale variabelen.

5.8 Functienaam overloading

In Java en C++ kunnen twee methoden in dezelfde naamruimte dezelfde naam hebben als hun argument types of aantal argumenten verschillend zijn. In C is dit niet het geval en alle functienamen dienen uniek te zijn!

5.9 Macro's

5.9.1 Concept

Bij gebruik van een macro wordt steeds tekstuele substitutie uitgevoerd van code. Een macro kan zonder en met argumenten (tussen haakjes ()) gedefinieerd worden. Er is een oppervlakkige gelijkenis met functies, maar door de tekstuele substitutie zorgen macro's voor een efficiëntere uitvoering (geen kopiëren van argumenten, springen naar het begin van een functie, na uitvoering de return-waarde kopiëren en terug springen naar de oproepende code).

Om het onderscheid te maken met functies, worden altijd hoofdletters gebruikt voor de naam van een macro.

5.9.2 Definitie

De syntax voor een macro-definitie is als volgt:

```
#define naam(arg1,arg2 ...) schrifttekst
```

Bemerkt dat er geen afsluitende ; is. Beschouw volgend voorbeeld van een macro-definitie en het gebruik van deze macro in vier verschillende gevallen:

```
#define SQUARE(x) x*x

double y=SQUARE(x);
double z=SQUARE(x+y);
double q=SQUARE(SQUARE(x+y));
double r=y/SQUARE(y);
```

Het eerste gebruik levert een correct resultaat op, de volgende drie toekenningen aan variabelen (**z**, **q** en **r**) leveren een ongewenst (foutief) resultaat!

Dit kan vermeden worden door volgende macro-definitie:

```
#define SQUARE(x) ((x)*(x))
```

Het bereik van een macro-definitie kan ook beperkt worden door gebruik van **#undef**:

```
#undef naam
```

vanaf deze regel kan de macro in dit bestand niet meer gebruikt worden.

Bemerk dat argumenten van macro's geen type hebben en dat macro-definities ook dikwijls in headerbestanden opgenomen worden (zoals functie declaraties). Sommige compilers laten toe resultaat van macro-expansie (na de pre-precessor) te zien, bijvoorbeeld door de compiler optie -E:

```
CC -E bestand.c
```

5.9.3 Macro's zonder argumenten

Deze worden zeer dikwijls gebruikt voor de definitie van constanten, zoals geïllustreerd wordt met volgende voorbeelden:

```
#define PI 3.1415926
#define SMALL 1E-20
#define EOF (-1)
#define MAXINT 2147483647 (overflowdetectie)
#define EOS ('\0')
#define EQUALS ==

if(i EQUALS j) ...
```

5.9.4 Gebruik van # bij macro's

Het gebruik van # bij een macro-definitie zorgt voor omzetting van een argument naar een string (en wordt daarom de "stringisatie-operator" genoemd):

#arg wordt omgezet naar "arg"

Volgende code illustreert het gebruik van de # operator:

```
/* stringization.c
 */
#include <stdio.h>
#define PRINTVAR(t,x) printf(" "#x " = %" #t" ",x)

int main(void) {
    double d=1.0;
    char c='a';
    d+=c;
    PRINTVAR(f,d);
    PRINTVAR(c,c);
    return 0;
}
```

Bij compilatie worden strings gescheiden door wit ruimte steeds naar één string omgezet.

5.9.5 Gebruik van ## bij macro's

Het gebruik van `##` bij een macro-definitie zorgt voor de concatenatie van twee argumenten tot één string (en wordt daarom de "token merger-operator" genoemd):

`token1##token2` wordt omgezet naar `"token1token2"`

Volgende code illustreert het gebruik van de `##` operator:

```
/* merger.c
 */

#include <stdio.h>
#define X(i) x##i
#define PRINT(i) printf("%d ",X(i))

int main(void) {
    int x1=10,x2=20,x3=30;
    PRINT(1);
    PRINT(2);
    PRINT(3);
}
```

5.10 Functies met willekeurig aantal argumenten

Een belangrijk voorbeeld van een functie met willekeurig aantal argumenten is de `printf` functie:

`printf(formaat_string,...)`

Hierbij geeft ... (het ellipsis symbool) aan dat men het aantal en het type van de argumenten niet op voorhand kent. Indien men zelf een dergelijke functie wil schrijven, dient men het headerbestand `<stdarg.h>` te includeren. In dit headerbestand wordt enerzijds een gegevensstructuur gedeclareerd en anderzijds 3 macro's gedefinieerd:

- `va_list`: is de gegevensstructuur die de argumentenlijst zal bevatten
- 3 macro's:
 1. `va_start`: zorgt voor initialisatie en vereist aantal argumenten (bijv. verkregen via laatst gekende argument),
 2. `va_arg`: verschaft toegang tot een argument (dient sequentieel opgeroepen te worden),
 3. `va_end`: dient opgeroepen te worden na toegang tot het laatste argument (opruimen van de gegevens).

Volgende beperkingen bij het schrijven van functies met variabele argumenten kunnen onderscheiden worden:

1. het aantal argumenten dient bepaald te worden. Dit kan bijv. via een gekend argument, waarin het aantal dient ingevuld te worden, of via een afspraak: een sentinel (of gekend afsluitargument) of een codering in het eerste argument (bijv. bij `printf` zorgt telling van het aantal % symbolen, verminderd met het aantal %%-symbolen, voor het aantal argumenten die volgen),
2. het type van de argumenten moet gekend zijn (of geëncodeerd zijn in een argument, zoals de conversiekarakters na % bij de `printf` functie),
3. de argumenten dienen steeds sequentieel doorlopen te worden.

Volgend code-voorbeeld illustreert een functie `gemiddelde`, die het gemiddelde berekent van een variabel aantal argumenten en waarbij de conventie gehanteerd wordt dat het eerste argument `n` het aantal volgende argumenten bevat.

```
/* ellipsis.c
*/
#include <stdio.h>
#include <stdarg.h>

double gemiddelde(int n,...);

int main(void) {
    printf("Gemiddelde = %f\n",gemiddelde(3,1,2,3));
    printf("Gemiddelde = %f\n",gemiddelde(2,1,2));
}

double gemiddelde(int n, ...) {
    int som=0;
    va_list arg;
    int i;
    va_start(arg,n);
    for(i=0;i<n;i++)
        som+= va_arg(arg,int) ;
    va_end(arg);
    return (double)som/n;
}
```

Het voorbeeld toont de declaratie van deze functie `gemiddelde`, samen met het oproepen in de `main` functie en de eigenlijke definitie van deze functie `gemiddelde`.

Hoofdstuk 6

C Preprocessor

In dit hoofdstuk wordt verder ingegaan op het gebruik van de C preprocessor (voor conditionele compilatie). Dit is enkel mogelijk in C (en C++), en niet in andere talen (zoals Java).

Elk C bronbestand wordt onderworpen aan een preprocessing fase met als voornaamste doelen: includering van headerbestanden, conditionele compilatie en macro expansie. Conditionele compilatie laat toe om code selectief te compileren, afhankelijk van een voorwaarde. Een belangrijk gebruik hiervan is het tegengaan van meervoudige declaraties, indien een headerbestand meer dan éénmaal geïncludeerd wordt. Daarom dienen headerbestanden met een `#if !defined #define ... #endif` constructie *beschermd* te worden, zoals in dit hoofdstuk uitgelegd wordt.

6.1 Werking preprocessor

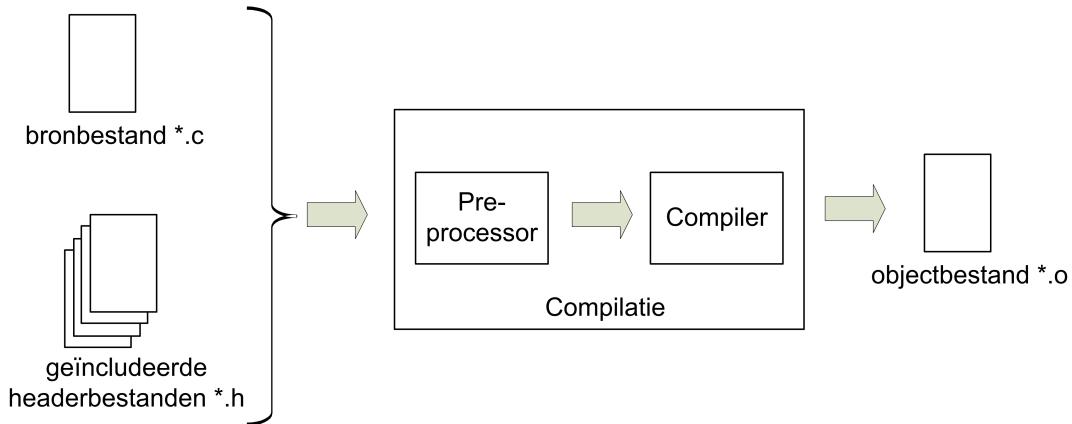
De preprocessor zorgt voor drie belangrijke functionaliteiten:

1. uitvoering van de `#include` opdrachten: toevoeging van de geselecteerde header-bestanden (i.e. deze letterlijk opnemen) in het te compileren bestand,
2. tekstuele substitutie van macro's (cfr. hoofdstuk 5), zodat de compiler de code met substituties aangeboden krijgt,
3. filtering van de code vooraleer deze door de compiler wordt behandeld. Als ontwikkelaar kan je dit filter-proces beïnvloeden aan de hand van conditionele compilatie, hetgeen hieronder uitgebreid behandeld wordt.

Figuur 6.1 situeert de werking van de preprocessor in het geheel van compilatie.

6.2 Conditionele compilatie

Hierbij worden condities in de code vermeld, die ervoor zorgen dat de bijhorende code enkel doorgegeven wordt aan de compiler als de condities voldaan zijn. De syntax voor conditionele compilatie is als volgt:



Figuur 6.1: Situering van de preprocessor in C

```

#ifndef <constante_gehele_uitdrukking>
#define  <naam>
#ifndef <naam>
...
code
...
#endif

```

waarbij één van de bovenste 3 regels gebruikt wordt (een dergelijke uitdrukking, die door de preprocessor geïnterpreteerd wordt, wordt een *directive* genoemd). De code in het midden wordt enkel aan de compiler doorgegeven indien (afhankelijk van de keuze van de bovenste 3 regels):

- <constante_gehele_uitdrukking> !=0
- <naam> gedefinieerd is
- <naam> niet gedefinieerd is

Verwante directieven zijn: `#else` en `#elif` (laatste komt overeen met `else if`). Alternatieve directieven (met dezelfde betekenis) zijn:

```

#define <naam>
#if defined <naam>
#endif(<naam>)

```

Ook logische operatoren `&&`, `||` en `!` zijn toegelaten. Het definiëren van een naam, gebeurt aan de hand van (cfr macro-definitie):

```
#define <naam>
```

en het ongedaan maken van een definitie (vanaf de huidige regel in een bestand) gebeurt aan de hand van:

```
#undef <naam>
```

Conditionele compilatie wordt in drie belangrijke gevallen aangewend, welke hieronder aan bod komen.

6.2.1 Faciliteit tijdens ontwikkeling

Aan de hand van conditionele compilatie kunnen bepaalde stukken code uitgefilterd worden bij het compileren, bijv. stukken code die nog niet klaar zijn of die debug-informatie afdrukken of naar een log-bestand schrijven. Volgend voorbeeld illustreert dit:

```
#define DEBUG

...
#ifndef DEBUG
printf("a= %d",a);
#endif
...
```

6.2.2 Portabiliteit

Code die ontwikkeld wordt om voor verschillende platformen te compileren (bijv. voor MAC, SUN Solaris of Intel PC) dient soms gebruik te maken van datastructuren of functies, die specifiek zijn voor een bepaald platform. De code dient dan alle mogelijkheden te voorzien en aan de hand van conditionele compilatie wordt voor het geselecteerde platform de juiste code uitgefilterd. Volgende code illustreert hoe aan de hand van conditionele compilatie het juiste (platform-specifieke) headerbestand wordt geïncludeerd:

```
#ifdef MAC
    #include <header1_mac.h>
#elif defined SUN
    #include <header1_sun.h>
#else
    #include <header1_pc.h>
#endif
```

Bovenaan het bestand of in een headerbestand dat bovenaan geïncludeerd wordt, dient men dan door een `#define` het gewenste platform te selecteren. Dit wordt hieronder geïllustreerd:

```
#define MAC
#include <header1.h>
#include <header2.h>
...
```

Combinatie van conditionele compilatie voor verhoging van portabiliteit en faciliteit tijdens de ontwikkeling, is uiteraard ook mogelijk, zoals onderstaande code illustreert:

```
#if defined(HP)&&(defined(DEBUG) || defined(TEST))
printf("Voorlopige versie voor HP");
#endif
```

6.2.3 Bescherming van headerbestanden

In headerbestanden worden vaak `#ifndef #define #endif` constructies toegevoegd. Beschouw volgend voorbeeld: een bestand `A.h` bevat een struct-definitie:

```
struct A {
    ...
};
```

In bestand `B.h` wordt deze struct ook gebruikt (waardoor `A.h` dient geïncludeerd te worden):

```
#include "A.h"
struct B {
    struct A a;
    ...
};
```

Wanneer in een ander bestand, bijv. `main.c` beide struct's gebruikt worden, dient men beide headerbestanden te includeren:

```
#include "A.h"
#include "B.h"
...
struct A a;
struct B b;
...
```

Bij compilatie van dit bestand zal de compiler een fout geven, waardoor compilatie mislukt: headerbestand `A.h` wordt tweemaal geïncludeerd, waardoor `struct A` tweemaal gedefinieerd wordt!

Om dit te vermijden wordt in het headerbestand `A.h` het volgende toegevoegd:

```
#ifndef _A_HEADER
#define _A_HEADER

struct A {
    ...
};

#endif
```

de naam `_A_HEADER` is hierbij willekeurig gekozen (de enige regel is dat geen bestaande naam gekozen wordt om overlap te vermijden). Het is een zeer goede aanpak in C om elk headerbestand met een dergelijke constructie te beschermen.

6.3 C preprocessor: voorgedefinieerde macro's

In tabel 6.1 worden voorgedefinieerde macro's weergegeven die door de preprocessor (net voor de compilatie) geëvalueerd worden.

Aan de hand van de `#line` directieve kan men de waarde van `__LINE__` en `__FILE__`

<code>__DATE__</code>	string met de datum
<code>__FILE__</code>	string met de bestandsnaam
<code>__LINE__</code>	geheel getal met huidig lijnnummer
<code>__STDC__</code>	ANSI C (1) of niet (!=1)
<code>__TIME__</code>	string met de tijd

Tabel 6.1: Overzicht voorgedefinieerde macro's

aanpassen. Dit is bijvoorbeeld handig als men met een beperkte editor werkt: door een lijnnummer in een bepaalde sectie van de code naar een mooi rond getal te plaatsen (bijv. 1000), zorgt men ervoor dat men bij een foutmelding op lijn 1012 onmiddellijk de juiste lijn terugvindt (lijn 372 is moeilijker terug te vinden als de editor dit niet supporteert). Analoog kan men door de bestandsnaam in verschillende secties telkens aan te passen, ervoor zorgen dat men op basis van de bestandsnaam bij een foutmelding direct de juiste sectie in een groot bestand kan lokaliseren.

Volgend voorbeeld illustreert het gebruik van een voorgedefinieerde macro om net vóór het compileren de nodige informatie in te vullen:

```
#include <stdio.h>
#define VERSION "Release 3.0"
#define DEBUG

void main() {
    short volatile timer;

    printf(" Release information: %s \n %s \n" , VERSION , __FILE__ );
    printf(" Modification on %s at %s \n" , __DATE__ , __TIME__ );

#ifdef DEBUG
    printf("DEBUG> at line %d: value of local timer : %d \n\n",
           __LINE__ , timer);
#endif
}
```

Bemerkt dus dat de waarde van een voorgedefinieerde macro bij het starten van de compilatie ingevuld wordt en dus bij uitvoering van het programma niet meer aangepast wordt.

6.4 assert macro

Deze macro biedt de mogelijkheid om een runtime (tijdens de uitvoering van het programma) controle op een expressie uit te voeren. Als de `assert` faalt, dan wordt de `abort()`-functie opgeroepen om het programma te stoppen.

Volgend voorbeeld illustreert het inlezen van een geheel getal en controle via `assert` of het ingelezen geheel getal wel tussen de gevraagde grenzen ligt:

```
int input=0;
printf("Enter integer in range ]0,10[: \n");
scanf("%d", &input);

assert(input > 0 && input <10);
```

6.5 Compile-time checks: #error

Naast runtime controles, is het soms ook handig om compile-time controles uit te voeren en de compilatie te beëindigen wanneer niet aan bepaalde voorwaarden voldaan is. Dit is mogelijk met de `#error` directieve. Het voorbeeld hieronder illustreert dit:

```
#ifdef MOTOROLA
    #error ERROR: Only Mac or Win32 targets supported!
#endif
```

Hoofdstuk 7

Rijen en Wijzers

In dit hoofdstuk komen enkele fundamentele concepten aan bod. Derhalve is dit een zeer belangrijk hoofdstuk. De Engelse vertaling van de titel luidt: *Arrays and Pointers*.

7.1 Rijen (Eng.:Arrays) in één dimensie

7.1.1 Concept

Rijen (Eng.: arrays) zijn een reeks van variabelen van HETZELFDE type (m.a.w. een rij of array is steeds homogeen), die naast elkaar (aanéénsuitend) in het geheugen zijn opgeslagen. De declaratie van een rij in C gaat als volgt:

```
type naam[constante_gehele_uitdrukking];
```

waarbij **naam** de naam van de rij is, **type** het type van de elementen uit de rij vastlegt, en **constante_gehele_uitdrukking** vastlegt hoeveel elementen de rij maximaal kan bevatten. Twee voorbeelden hiervan zijn:

```
double neerslag[12];
long postnummer[100];
```

Declaraties gebeuren ook vaak in combinatie met een macro, om de lengte van de rij vast te leggen (op deze manier kan de lengte van de rij achteraf gemakkelijk aangepast worden). Het voorbeeld hieronder illustreert dit:

```
#define MAANDEN 12
...
double neerslag[MAANDEN];
```

Het gebruik van een rij gebeurt aan de hand van volgende uitdrukking:

```
naam[gehele_uitdrukking]
```

waarbij $0 \leq \text{gehele_uitdrukking} < \text{constante_gehele_uitdrukking}$. Het is dus belangrijk dat `gehele_uitdrukking` NOoit $\geq \text{constante_gehele_uitdrukking}$.

De waarde van `gehele_uitdrukking` wordt ook de *index* van de rij genoemd. De uitdrukking `naam[gehele_uitdrukking]` kan overal gebruikt worden, waar een gewone variabele van hetzelfde type toegelaten is. Bemerkt dat `[]` een C-operator is, waarbij de precedentieregels uit tabel 4.4 gelden.

7.1.2 Initialisatie

Volgende syntax (in BNF notatie) dient aangewend te worden voor initialisatie van een rij:

```
type naam[constante_gehele_uitdrukkingopt] =
    { constante_uitdrukking {, constante_uitdrukking}0+};
```

Twee zaken vallen hierbij op:

- niet alle elementen hoeven geïnitialiseerd te worden: de niet geïnitialiseerde waarden worden automatisch op 0 geplaatst!
- de lengte van een rij hoeft niet opgegeven te worden (is optioneel): indien de lengte (ook wel het bereik genoemd) niet opgegeven is, vult de compiler zelf het aantal elementen in .

Ter illustratie, bij volgende declaratie:

```
double neerslag[12] = {20.7,23.0,99.0,77.4};
```

krijgen de eerste 4 elementen een waarde en de volgende 8 elementen worden op 0.0 geplaatst. Volgende twee declaraties zijn ook equivalent:

```
int a[4]={1,2,3,4};
int a[ ]={1,2,3,4};
```

Belangrijk is dat er geen automatische initialisatie is: indien dit gewenst of aangewezen is, dient men steeds tussen accolades een initialisatie te specifiëren.

7.1.3 Karakter-rijen

Voor karakter-rijen zijn volgende twee initialisaties equivalent:

- aan de hand van een string,
- opsomming tussen {} van de individuele karakters, belangrijk is om het '\0' als afsluiter in dit geval niet te vergeten (komt aan bod in sectie 7.5).

Het voorbeeld hieronder illustreert dit:

```
char a[ ] = "Een tekst";
char a[ ] = {'E','e','n',' ', 't','e','k','s','t','\0'};
```

7.1.4 C-idioom

Voor het behandelen van de elementen uit een rij worden in C vaak for-lussen gebruikt. Een typische constructie in C wordt hieronder getoond:

```
#define N 100

...
int a[N];
int i;
...
for(i=0;i<N;i++)
/* bewerk element a[i] */
```

7.1.5 Arrays als functie-argumenten

Array's kunnen als functie-argument doorgegeven worden. Het argument bij declaratie ziet er dan uit als volgt:

```
type [ ]
of: type naam[ ]
```

en bij definitie van de functie uiteraard:

```
type naam[ ]
```

vermits **naam** verder gebruikt wordt in de implementatie (tussen {}). Volgend voorbeeld illustreert dit:

```
/* berekensom.c */

#include <stdio.h>

int bereken_som(int[ ],int);

int main(void) {
    int a[ ]={1,2,3,4,5};
    printf("Som = %d\n",bereken_som(a,5));
    return 0;
}

int bereken_som(int a[ ],int n) {
    int som=0;
    int i;
    for(i=0;i<n;i++) som+=a[i];
    return som;
}
```

Bemerk dat de lengte van een array steeds expliciet dient meegegeven te worden: er bestaat geen `length()` functie of attribuut (zoals in Java) om dit op te vragen.

7.2 Wijzers (Eng.:Pointers)

7.2.1 Concept

Alle variabelen van niet-primitieve types in Java zijn referenties. C heeft geen concept van referentie, maar wel van wijzers, welke in Java niet aanwezig zijn.

Een wijzer is een variabele, die een adres bevat: op dit adres is data opgeslagen, ofwel een individuele variabele ofwel een rij van variabelen. In dit laatste geval bevat de wijzer typisch het adres van het eerste element, en vermits elementen van rijen steeds aanéénsluitend in het geheugen zijn opgeslagen, kan men het adres van elk element in de rij eenvoudig uitrekenen.

7.2.2 Operatoren & en *

De `&`-operator, toegepast op een variabele, levert het adres van deze variabele, terwijl de `*`-operator, toegepast op een wijzer-variabele, de waarde geeft van de data waarvan de wijzer-variabele het adres bevat. Via de `*`-operator kan een andere waarde aan de data toegekend worden.

7.2.3 Wijzertypes

Voor elk type is er een wijzertype. Voor het `int` type, wordt het wijzer-naar-`int` type voorgesteld door `int*`. `float*` is het wijzer-naar-`float` type. In het algemeen geldt dat elk type een geassocieerd wijzertype heeft, namelijk `type*`.

Bij het toekennen van een wijzerwaarde aan een variabele of het vergelijken van twee pointerwaarden, dienen de types overeen te komen.

7.2.4 NULL waarde

Een geldige waarde voor een wijzer is `NULL` (i.e. de wijzer-variabele bevat allemaal nullen, alle bits op 0), hetgeen betekent dat de wijzer nergens naar verwijst. Pas dus nooit de `*` operator op een `NULL` wijzer toe !! `NULL` is als macro gedefinieerd in de standaardbibliotheek (Sectie 7.6).

Er kan eenvoudigweg getest worden of een wijzervariabele `p` verschillend is van 0 door de expressie:

```
if (!p) or if (p != NULL).
```

Indien een wijzer-variabele niet geïnitialiseerd werd of geen geldig adres werd toegekend, kan het overall naar wijzen of kan het toevallig `NULL` zijn. Een dergelijke wijzer wordt een zwevende wijzer (Eng.: dangling pointer) genoemd. Pas nooit de `*`-operator toe op een ongedefinieerde wijzer !!!!

7.2.5 Conversie-karakter

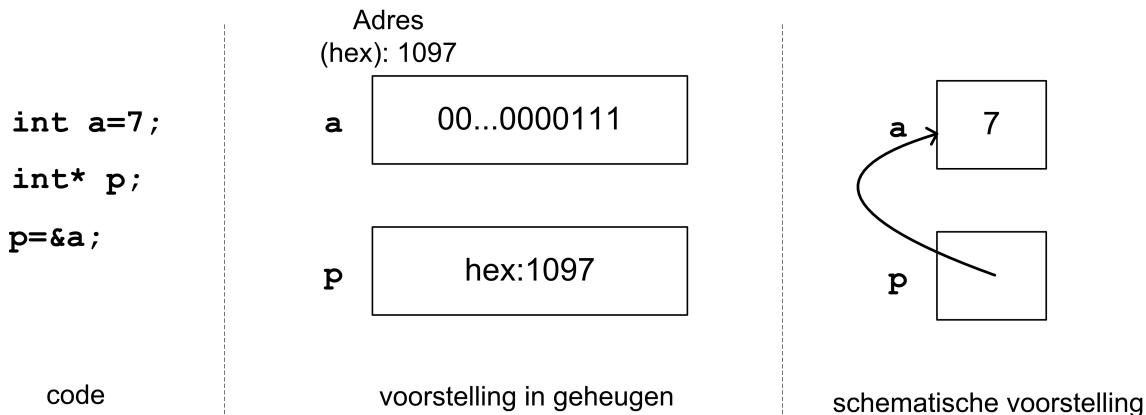
Voor output via `printf` is er een conversie-karakter `p` gedefinieerd om de inhoud van een wijzervariabele (i.e. het adres dat de variabele bevat) in hexadecimale voorstelling af te drukken. Een voorbeeld hiervan is:

```
printf("Wijzer = \"%p\",p);
```

7.2.6 Voorbeeld

```
int a=7;
int* p= NULL;
int b;
p=&a;
a=9;
*p=12;
b = *p;
```

In dit voorbeeld wordt geïllustreerd dat een wijzer-variabele `p` het adres van een `int`-variabele `a` bevat, en de variabele `a` via `*p` een waarde krijgt. Tenslotte krijgt `b` dezelfde waarde als `a`. Figuur 7.1 illustreert de voorstelling in het geheugen en de schematische voorstelling ervan. In het voorbeeld hieronder wordt een wijzer-variabele `p` gebruikt om



Figuur 7.1: De voorstelling in het geheugen van een wijzervariabele en de schematische voorstelling ervan.

het adres van elementen uit een array te bevatten en de array-elementen incrementerend een waarde te geven:

```
#include <stdio.h>
#define N 5

int main(void) {
    int a[N]={0};
    int* p=0;
```

```

int i;
for(i=0;i<N;i++) {
    p=&a[i];
    printf("wijzer naar a[%d] : 0x%p\n",i,p);
    *p=i;
}
for(i=0;i<N;i++)
    printf("a[%d]=%d\n",i,a[i]);
return 0;
}

```

7.2.7 Relationale operatoren

De relationele operatoren kunnen op wijzer-variabelen toegepast worden: de numerieke adressen (i.e. de inhoud van de wijzervariabelen) worden dan met elkaar vergeleken.

7.2.8 Meervoudige wijzers

Een wijzer-variabele kan verwijzen naar een variabele, die zelf een wijzervariabele is: op die manier ontstaat een dubbele wijzer. Drie- of vierdubbele wijzers zijn ook mogelijk. Volgende code illustreert dit:

```

int a = 7;
int* p = &a;
int** q = &p;
printf("p= %p, a= %d",*q,**q);

```

7.2.9 Pointer arithmetica

Hiermee worden rekenkundige bewerkingen op wijzervariabelen bedoeld. Op het adres van array elementen kan simple arithmetica toegepast worden. De incrementatie van een adres met 1 doet het verwijzen naar het volgende element in de array. Vermindering van het adres met 1 doet het verwijzen naar het vorige element. Analoog, bij vermeerdering met **n** verwijst de variabele naar het **n**-volgende element, en bij vermindering met **n** naar het **n**-vorige element. De code hieronder:

```

int* p=(int*)1000;
p++;
p--;
p+=2;
p-=4;

```

is dus equivalent met:

```

1000+sizeof(int)
1000-sizeof(int)
1000+2*sizeof(int)
1000-4*sizeof(int)

```

Het effect van een rekenkundige bewerking op een wijzervariabele hangt dus af van de lengte van het type (i.e. het aantal ingenomen bytes in het geheugen), te bepalen via de `sizeof` operator!

7.2.10 Generieke wijzer-types

Het is soms noodzakelijk om een wijzer-variabele op te slaan of door te geven zonder het type te kennen van de data waar hij naar verwijst. Hiervoor kan een wijzer-variabele van het type `void*` gebruikt worden. Een dergelijke wijzer-variabele kan steeds aan een ander type wijzer-variabele toegekend worden. Op een generieke wijzer-variabele kan de `*` operator niet toegepast worden en kan ook geen pointer arithmetic toegepast worden. Het is de verantwoordelijkheid van de ontwikkelaar om ervoor te zorgen dat de pointer waarde als het correcte type geïnterpreteerd wordt.

Generieke pointers zijn noodzakelijk voor dynamisch geheugenbeheer (cfr Sectie 7.4).

In het voorbeeld hieronder wordt getoond hoe men via `void*` de assignatie van wijzer-variabelen van een verschillend type toch kan bewerkstelligen.

```
int *i;
double *d;

i=d; /* mogelijk in traditionele C, niet in ANSI C */

int *i;
double *d;
void* v;

i=d;
i=v=d; /* mogelijk in traditionele C, ook in ANSI C! */
```

7.2.11 Verband array's en pointers

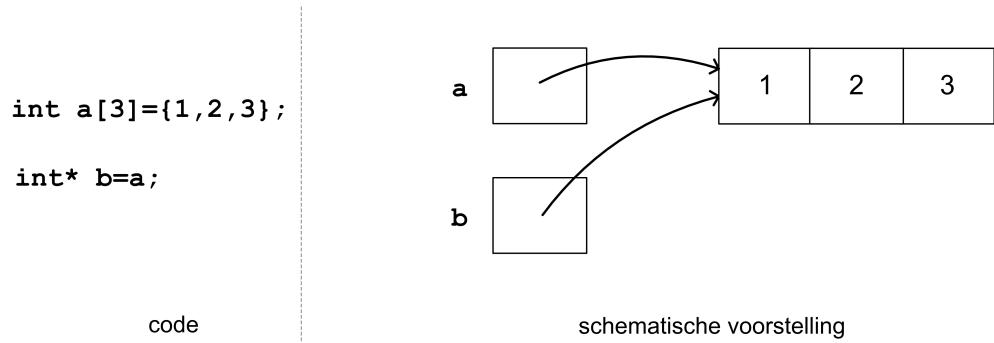
Een belangrijke conventie is dat de array-naam overeenkomt met een wijzervariabele, die het adres van het eerste element bevat. De waarde van deze variabele kan in programma-code *niet* aangepast worden.

Merk op dat het de verantwoordelijkheid is van de programmeur om de lengte van de array te specificeren ofwel de conventie te gebruiken dat het laatste element van de array een speciaal karakter is (meestal wordt '0' als speciaal karakter genomen). Een veelgemaakte fout is dat de `sizeof` operator gebruikt wordt voor evaluatie van de lengte van een array: dit levert enkel de grootte in bytes op van de pointer naar het eerste element!

Het verband tussen array's en pointers wordt geïllustreerd in volgend voorbeeld:

```
int a[3]={1,2,3};
int *b=a;
```

Figuur 7.2 toont dit schematisch.



Figuur 7.2: Schematische voorstelling in het geheugen van een array en een wijzer naar de array.

b bevat dan het adres van het eerste element van de array **a**. Aanpassingen van **a** is niet mogelijk, **b** kan wel aangepast worden:

```
a++; /* foutmelding door compiler*/
b++; /* geen probleem: b bevat dan adres van volgende element in array */
```

In plaats van **a[i]** kunnen ook volgende equivalenten uitdrukkingen gebruikt worden:

```
a[i]
*(a+i)
*(b+i)
```

Vraag: Wat is de output van volgende code? Leg uit.

```
/* verschil.c */
#include <stdio.h>

int main(void) {
    float a[]={1.0,2.0,3.0,4.0,5.0};
    float*start,*eind;
    int i=0;

    start=a;
    eind=&a[4];
    for(i=0;i<5;i++)
        printf("a[%d]=%f\n",i,*(start+i));
    printf("eind - start = %d\n",eind-start);
    printf("(int)eind-(int)start = %d\n", (int)eind-(int)start);
    return 0;
}
```

7.2.12 Oefening

Hieronder wordt een voorbeeld getoond, waarbij arrays en wijzers gecombineerd worden. Het programma bevat echter 2 fouten. Welke?

```
#include <stdio.h>
#define SIZE 10

int main(void) {
    int a[SIZE];
    int *p;
    for(p=&a[0]; p<&a[SIZE];)
        *++p=0;
}
```

Vraag: Is er een verschil tussen beide onderstaande declaraties? Zo ja, wat is het verschil?

```
int *a[10];
int (*a)[10];
```

7.3 Call by reference

7.3.1 Principe

In Java worden alle primitieve types *by value* als functie-argument doorgegeven, i.e. de functie krijgt enkel de waarde en kan de originele waarde niet aanpassen. Alle referentietypes worden *by reference* als functie-argument doorgegeven: de functie kan de publieke inhoud van het gerefereerde object aanpassen.

In C worden alle types *by value* doorgegeven en de waarde van argumenten kan dus niet veranderd worden (enkel de lokale kopie van de variabelen in het functie-blok kan aangepast worden). Als echter een wijzer naar een variabele doorgegeven wordt, kan de functie de *-operator op zijn kopie van de wijzer toepassen en dus de waarde van de variabele aanpassen (dit wordt *pass by reference* genoemd).

Samenvattend, indien men wenst dat de waarde, doorgeven aan een functie, in de opropende context kan gewijzigd worden, dient men te zorgen voor het volgende:

- Geef de adressen van variabelen door.
- De formele parameters zijn variabelen van een wijzertype.
- Gebruik dereferentie (*-operator) van parameters om de waarde van de variabelen uit de opropende context te wijzigen.

Volgende code illustreert dit, zonder doorgeven van adressen:

```
...
char a;
opvolger(a);
... /* waarde a is niet aangepast */
void opvolger(char a) {
    a++;
}
```

met doorgeven adressen:

```

...
char a;
opvolger(&a);
... /* waarde a is wel aangepast */
void opvolger(char* a) {
    (*a)++;
}

```

7.3.2 Voorbeeld

Onderstaande functie `wissel` werkt niet correct: corrigeer!

```

/* wisselnaief.c */

#include <stdio.h>

void wissel(int,int);

int main(void) {
    int a=1,b=2;
    printf("a=%d b=%d\n",a,b);
    wissel(a,b);
    printf("a=%d b=%d\n",a,b);
    return 0;
}

void wissel(int a,int b) {
    int t=a;
    a=b;
    b=t;
}

```

De correcte versie wordt voor de volledigheid hieronder weergegeven (bestudeer zorgvuldig de verschillen):

```

/* wisselints.c */

#include <stdio.h>

void wissel(int*,int*);

int main(void) {
    int a=1,b=2;
    printf("a=%d b=%d\n",a,b);
    wissel(&a,&b);
    printf("a=%d b=%d\n",a,b);
    return 0;
}

void wissel(int *a,int *b) {

```

```

int t=*a;
*a=*b;
*b=t;
}

```

7.3.3 Access type modifiers

Bij declaratie van variabelen, kan men twee sleutelwoorden gebruiken om de toegang tot de variabelen te specifiëren: **const** en **volatile**. Deze worden nu in detail besproken.

const

Het sleutelwoord **const** verhindert de wijziging van een variabele. Uiteraard is initialisatie in dit geval verplicht!

Via een wijzervariabele kan men echter WEL een **const** variabele aanpassen, zoals volgende voorbeeld toont:

```

int main(void) {
    const int i=7;
    int* p=&i;
    *p=12; /* warning! */
    printf("i=%d\n",i);
    return 0;
}

```

Dit levert een waarschuwing (Eng.: warning) op door de compiler, maar het programma wordt wel gecompileerd. Wanneer men echter het **const** sleutelwoord gebruikt bij declaratie van de wijzervariabele, dan lukt de compilatie niet meer:

```

int main(void) {
    const int i=7;
    const int* p=&i;
    *p=12; /* error */
    printf("i=%d\n",i);
    return 0;
}

```

De compiler geeft een foutmelding en stopt de compilatie.

Er zijn twee mogelijkheden om **const** te gebruiken in combinatie met wijzer-variabelen:

1. **const int *p;** : de waarde van de variabele waar p naar wijst is constant (***p** is constant)
2. **int const *p;** : de variabele p is constant (p is constant)

const in functie prototype

Gebruik van **const** in een functie prototype verhindert de (onopzettelijke) wijziging van call-by-reference argumenten. Volgende code illustreert dit:

```
#include <stdlib.h>

void opvolger(const char*);

int main(void) {
    char a='a';
    printf("a=%c\n",a);
    opvolger(&a);
    printf("a=%c\n",a);
    return 0;
}

void opvolger(const char* a) {
    (*a)++; /* error */
}
```

De compiler geeft in dit geval een foutbericht op de regel `(*a)++;` omdat `*a` volgens het prototype niet aangepast mag worden.

volatile

Een variabele kan onverwacht van waarde veranderen. Voorbeelden hiervan zijn:

1. hardware registers voor communicatie met een randapparaat,
2. variabelen die vanuit een interrupt service routine gewijzigd worden,
3. gedeelde variabelen in een multitasking applicatie.

Het **volatile** geeft aan de compiler aan dat de waarde van deze variabelen telkens moet opgevraagd worden, wanneer ze nodig zijn (omdat ze onverwachts kunnen aangepast worden).

Oefening

Vragen:

1. Kan een parameter zowel **const** als **volatile** zijn ?
2. Kan een pointer **volatile** zijn ?
3. Wat is het probleem met volgende functie:

```
int square (volatile int* ptr) {
    return *ptr * *ptr;
}
```

7.3.4 Array als functie-argument: gebruik const

Arrays worden aan functies by reference doorgegeven, i.e. het adres van de array wordt doorgegeven. In de parameterlijst zijn array- en wijzernotatie dus equivalent. Indien men wenst dat een array niet kan aangepast worden, dient men **const** te gebruiken. Hieronder worden twee equivalente functie prototypes getoond: in array-notatie en in wijzernotatie:

```
int grootste(int[],int);
...
int grootste(int a[],int n) {
    int i=0;
    int m=*a++;
    for(i=1;i<n;i++)
        m=(*a++>m) ? *(a-1):m;
    return m;
}

int grootste(int*,int);
...
int grootste(int* a,int n) {
    int i=0;
    int m=*a++;
    for(i=1;i<n;i++)
        m=(*a++>m) ? *(a-1):m;
    return m;
}
```

Het **const** sleutelwoord kan als volgt gebruikt worden:

```
int grootste(const int*,int);
...
int grootste(const int* a,int n) {
    int i=0;
    int m=*a++;
    for(i=1;i<n;i++)
        m=(*a++>m) ? *(a-1):m;
    return m;
}
```

waardoor men er zeker van is dat de array niet kan aangepast worden in de functie (zoniet geeft de compiler een foutbericht).

7.4 Dynamisch geheugenbeheer

Soms is het noodzakelijk om tijdens de uitvoering van een programma geheugen te kunnen reserveren en te kunnen vrijgeven. Dit wordt dynamisch geheugenbeheer genoemd (in tegenstelling tot statisch geheugenbeheer, wanneer men een vaste -tijdens het compilatieproces gekende- hoeveelheid geheugen reserveert). Dynamisch geheugenbeheer is in

Java voorziet via het `new` sleutelwoord en de *garbage collector* zorgt voor vrijgave van het geheugen als het niet meer nodig is. In C is dynamisch geheugenbeheer aanwezig door functies in `<stdlib.h>`. Deze worden hieronder één voor één toegelicht.

7.4.1 Toewijzen van geheugen - memory allocation

Dit betekent de reservatie van geheugen en het verkrijgen van een wijzer naar het gereserveerde geheugen. Volgende twee functies dienen hiervoor gebruikt te worden:

```
void* malloc(size_t aantal_bytes);
void* calloc(size_t aantal, size_t aantal_bytes);
```

Het type `size_t` is een typedef van `unsigned int`. De functie `malloc` geeft een pointer terug naar de start van een geheugenblok dat net voldoende groot is om `aantal_bytes` bytes op te slaan. Het geeft een generieke wijzer (`void*`) terug, die aan een wijzer van elk type kan toegekend worden. Gealloceerd geheugen via `malloc` is niet geïnitialiseerd. De functie `calloc` initialiseert het geheugen wel (alle waarden op 0).

Om de vereiste hoeveelheid geheugen te bepalen kan men de grootte opvragen van een bepaald type gebruikmakend van de operator `sizeof(type)` (dit zorgt voor systeem-onafhankelijke code). Voor een array dient dit dan vermenigvuldigd te worden met de vereiste grootte van de array.

Bij de functie `calloc` zijn er twee argumenten: het aantal elementen en de grootte (in bytes) van één element.

Belangrijk is dat de beide functies `NULL` teruggeven als de reservatie van geheugen mislukt is (onvoldoende geheugen).

7.4.2 Vrijgeven van geheugen - releasing memory

In Java blijft een object bestaan zolang er een referentie naar is. In C kan data in het geheugen verwijderd worden zelfs als er verschillende wijzers naar zijn (door de `free`-functie), de ontwikkelaar is volledig verantwoordelijk om ervoor te zorgen dat wijzers een geldig adres bevatten wanneer ze gebruikt worden.

Al het gealloceerde geheugen, aangemaakt met `malloc`, `calloc` en `realloc`, moet vrijgegeven (Eng.: released) worden wanneer het niet langer vereist is door de start-wijzer door te geven als argument aan de `free`-functie. Enkel wijzerwaarden teruggekregen van `malloc`, `calloc` of `realloc` kunnen doorgegeven worden aan de `free`-functie.

De syntax van de functie `free` is als volgt:

```
void free(void* toegewezen_pointer);
```

Hierbij treedt er tijdens de uitvoering van een programma een fout op wanneer:

1. het argument `toegewezen_pointer` al vrijgegeven is,
2. het argument `toegewezen_pointer` een andere waarde heeft dan een toegewezen pointer, i.e. dan een returnwaarde van `malloc`, `calloc` of `realloc`.

Het niet vrijgeven van geheugen wordt een geheugenlek (Eng.: memory leak) genoemd: deze zijn soms moeilijk op te sporen. Voor details wordt verwezen naar de oefeningenlessen.

7.4.3 Hergebruik van geheugen - reuse of memory

De functie `realloc`:

```
void* realloc(void* toegewezen_pointer, size_t aantal_bytes);
```

poogt een bestaand gereserveerd geheugenblok (waar de variabele `toegewezen_pointer` naar wijst) te hergebruiken voor `aantal_bytes`. Indien `aantal_bytes` < de oorspronkelijke grootte van het geheugenblok, dan gaat het om een inkrimping, welke zeker lukt: de inhoud van het begin van het blok blijft behouden (de return waarde is identiek aan `toegewezen_pointer`). Wanneer `aantal_bytes` > de oorspronkelijke grootte van het geheugenblok, dan gaat het over een uitbreiding, welke niet noodzakelijk lukt. Men kan in dit geval 3 mogelijkheden onderscheiden:

1. de uitbreiding lukt op dezelfde plaats in het geheugen (i.e. achter het oorspronkelijke geheugenblok is voldoende vrije ruimte voor uitbreiding): de return waarde is dan identiek aan het argument `toegewezen_pointer`.
2. de uitbreiding lukt niet op dezelfde plaats in het geheugen (i.e. achter het oorspronkelijke geheugenblok is onvoldoende vrije ruimte voor uitbreiding), maar op een andere plaats lukt het wel: de inhoud van het oorspronkelijke geheugenblok wordt gekopieerd naar een nieuwe plaats en de return waarde van `realloc` is een nieuwe waarde, verschillend van het argument `toegewezen_pointer`.
3. de uitbreiding lukt niet op dezelfde plaats in het geheugen, en ook op een andere plaats lukt het niet: de return waarde van `realloc` is dan `NULL`.

7.4.4 Dynamisch geheugenbeheer: fouten en problemen

Dynamisch geheugenbeheer is noodzakelijk in C, maar is de oorzaak van veel fouten en problemen bij de uitvoering van programma's. We kunnen volgende onderscheiden:

- *zwevende pointer* (Eng.: dangling pointer): kan voorkomen wanneer een pointer nog gebruikt wordt na een `free`-oproep of wanneer meerdere pointers wijzen naar hetzelfde blok en dus na één `free` allen ongeldig worden.
- *geheugenlek* (Eng.: memory leak): kan voorkomen wanneer er geen pointers meer zijn naar het gereserveerde geheugen en er dus ook geen `free`-functie kan opgeroepen worden.
- *geheugen-fragmentatie* (Eng.: memory fragmentation): na veelvuldig alloceren en deallocceren kunnen bepaalde kleine stukjes geheugen niet meer gebruikt worden omdat ze te klein zijn.

In Java levert dynamisch geheugenbeheer geen problemen op, wegens de aanwezigheid van de *garbage collector*.

BELANGRIJK: er dient in C onderscheid gemaakt te worden tussen segment-geheugen (functie code en bibliotheek functies), stack-geheugen (lokale variabelen in functies en functie-argumenten), en heap-geheugen (dynamisch gealloceerde variabelen, globale variabelen en statische variabelen). Appendix A bevat verdere uitleg en illustratieve voorbeelden van correct dynamisch geheugenbeheer en veelgemaakte fouten.

7.4.5 Generieke code dankzij dynamisch geheugenbeheer

Het is handig om functies te kunnen schrijven die geldig zijn voor elk type en dus niet specifiek zijn voor één type. Beschouw als voorbeeld een functie **wissel**, waarbij men de waarden van de argumenten wil omwisselen. Door gebruik te maken van het type **void*** is de declaratie van de functie onafhankelijk van het specifieke type. Het derde argument van de functie geeft de grootte in bytes van het type dat men doorgeeft in de eerste twee argumenten.

```
/* wisselgeneriek */

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

void wissel(void*,void*,int);

int main(void) {
    int a=1,b=2;
    double c=3.14,d=2.71828;
    printf("a=%d b=%d\n",a,b);
    wissel(&a,&b,sizeof(a));
    printf("a=%d b=%d\n",a,b);
    printf("c=%f d=%f\n",c,d);
    wissel(&c,&d,sizeof(c));
    printf("c=%f d=%f\n",c,d);
    return 0;
}
```

Voor de implementatie van de functie dient men gebruik te maken van dynamische geheugenbeheer (omdat men op voorhand de waarde van **n**, het derde argument, niet kent) en de functie **memcpy** uit **<string.h>**:

```
void wissel(void *a,void *b,int n){
    void* temp= malloc(n);
    memcpy(temp,a,n);
    memcpy(a,b,n);
    memcpy(b,temp,n);
    free(temp);
```

```
}
```

`memcpy(a,b,n)` kopieert hierbij `n` bytes vanaf adres `b` naar adres `a`.

7.5 Strings

7.5.1 Concept

Een Java variabele van het type `char` kan elk Unicode karakter bevatten. In C, kan het `char` type elk karakter voorstellen in een karakter-verzameling, die afhangt van het type van systeem of platform waarvoor het programma gecompileerd is.

Java strings zijn objecten van de klasse `java.lang.String` of `java.lang.StringBuffer` en stellen sequenties van `char`'s voor. Strings in C zijn ofwel array's van `char`'s ofwel pointers naar `char`'s (i.e. een pointer naar het eerste element van de string). Functies die strings bewerken veronderstellen steeds dat de string getermineerd wordt met het null character '`\0`' (ook wel de schildwacht of Eng.: sentinel genoemd). Hiervoor kan ook de macro `EOS` (Eng.: End of String) gebruikt worden:

```
#define EOS '\0'
```

De vereiste geheugenruimte voor een string is steeds gelijk aan:

```
(aantal_karakters+1)*sizeof(char)
```

Een string of karakter array kan geïnitialiseerd worden zoals andere array's, bij gebruik van stringconstanten tussen "", voegt de compiler zelf '`\0`' toe.

7.5.2 Conversie-karakter

Het conversie-karakter voor strings bij `scanf` en `printf` is `s`, zoals hieronder geïllustreerd:

```
char *t="een tekst";  
printf("t bevat = %s",t);
```

Nuttige hulpfuncties voor manipulatie van strings zijn gedefinieerd in `<string.h>`, en worden hieronder toegelicht.

7.5.3 String versus char []

De code hieronder illustreert het verschil tussen gebruik van `char*` en `char[]` om strings voor te stellen:

```
char* s= "abc";  
char a[ ] = {'a','b','c','\0'};  
  
#include <stdio.h>
```

```

int main(void) {
    char a[ ]={'a','b','c','\0'};
    char *s="abc";
    printf("%c %c\n",a[0],s[0]);
    printf("%c %c\n",*(a+1),*(s+1));
    /* a+=2; aanpassing a niet toegelaten */
    s+=2;
    printf("%c %c\n",*(a+2),*s);
    return 0;
}

```

7.5.4 String functies

Volgende functies uit `<string.h>` zijn zeer nuttig voor string-bewerkingen:

- concateneren: `char *strcat(char *s1, const char *s2);`
- vergelijken: `int strcmp(const char *s1,const char *s2);`
- kopiëren: `char *strcpy(char *s1,const char *s2);`
- lengte bepalen: `size_t strlen(const char *s); /* exclusief EOS */`

Belangrijk is dat deze functies GEEN geheugen alloceren: dit dient door de ontwikkelaar apart te gebeuren! De gebruiker van de functies moet zorgen voor voldoende ruimte.

7.5.5 Geformatteerde I/O van/naar string

In `<stdio.h>` bevinden zich volgende functies

- invoer: `int sscanf(const char* s, const char* ctrl, ...);` uit string `s` worden gegevens gelezen en opgeslagen in de argumenten,
- uitvoer `int sprintf(char *s, const char* ctrl, ...);` naar string `s` worden gegevens geschreven volgens het formaat gespecificeerd in de string `ctrl`.

Belangrijk is dat allocatie van geheugen voor de argumenten of voor string `s` niet binnen `sscanf` of `sprintf` gebeurt, maar dat dit apart dient te gebeuren!

7.5.6 Voorbeeld: vergelijken van strings

Het werken met strings wordt hieronder geïllustreerd aan de hand van een programma waarin twee string's met elkaar vergeleken worden.

```

/* compare.c */
#include <stdio.h>
#include <string.h>

```

```

int my_strcmp(const char*,const char*);

int main(void) {
printf("gelijk = %d\n",my_strcmp("abcd","abcd"));
printf("niet gelijk = %d\n",my_strcmp("abcd","abcD"));
return 0;
}

int my_strcmp(const char* s1,const char*s2) {
if (strlen(s1)!=strlen(s2)) return 0;
else
while(*s1)
if((*s1++)!=(*s2++)) return 0;
return 1;
}

```

7.5.7 Belangrijk - Vermijden van buffer overflows

Het gebruik van bovenstaande functies `strcat`, `strcpy`, `sscanf` en `sprintf` kan resulteren in *onveilige* code (i.e. buitenstaanders kunnen ongewenst controle krijgen over het computersysteem, of gebruikers kunnen meer privileges, zoals admin toegang of root toegang, krijgen dan gewenst of denial-of-service attacks kunnen uitgevoerd worden om een applicatie te doen crashen). De reden hiervoor is dat in de code buffers aangemaakt worden, en er bij het invullen van de buffer geen controle is om buffer overflow (het overlopen van de buffer) te detecteren, of te vermijden.

De functies `strcat`, `strcpy`, `sscanf` en `sprintf` controleren inderdaad niet of de destinatiestring voldoende groot is om de copieer-operatie te kunnen doorvoeren. Er worden twee types van buffer overflow onderscheiden: (i) stack-gebaseerde buffer overflow en (ii) heap-gebaseerde buffer overflow. Zoals hierboven beschreven (en ook in Appendix A) wordt de *stack* gebruikt als geheugen bij het uitvoeren van functies en wordt de *heap* gebruikt voor het dynamisch alloceren van geheugen.

Stack-gebaseerde buffer overflow

In volgend programma wordt het principe geïllustreerd:

```

#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    char buff[100];
    if(argc <2){
        printf("Syntax: %s <input string>\n", argv[0]);
        exit (0);
    }
    strcpy(buff, argv[1]);
    return 0;
}

```

Wanneer het argument meer dan 100 karakters bevat, zal de geheugenruimte naast de variabele **buff** ook overschreven worden. Dit kan resulteren in:

- ofwel een crash van het programma omdat buiten de toelaatbare geheugenruimte van het programma geschreven wordt (een voorbeeld van een *denial-of service* aanval),
- ofwel uitvoering van ongewenste code, i.e. uitvoering van code die door een *hacker* nauwkeurig is doorgegeven bij het overschrijven de buffer. Deze geïnserteerde code kan zorgen voor installatie van een *trojan* programma, data corruptie of verkrijgen van **root** of **admin** toegang (wanneer bijvoorbeeld de code hierboven als root of admin uitgevoerd wordt).

Figuur 7.3 toont het principe van stack-gebaseerde buffer overflow en uitvoering van ongewenste code. Op de stack worden naast de functieargumenten, de return waarde, de lokale veranderlijken van een functie ook het *return adres* van de functie bijgehouden (i.e. het adres van de instructie in de oproepende code, die onmiddellijk na de beëindiging van de functie-oproep wordt uitgevoerd). Het gebruik van een stack voor oproep van functies komt meer uitgebreid aan bod in de cursus *Computerarchitectuur*.

Wanneer bij een buffer overflow het return adres doelbewust overschreven wordt en het adres bevat van een door de hacker ingevoegd code fragment (bijvoorbeeld aanwezig verder in de overschreven buffer), dan wordt dit code fragment uitgevoerd bij het beëindigen van de functie.

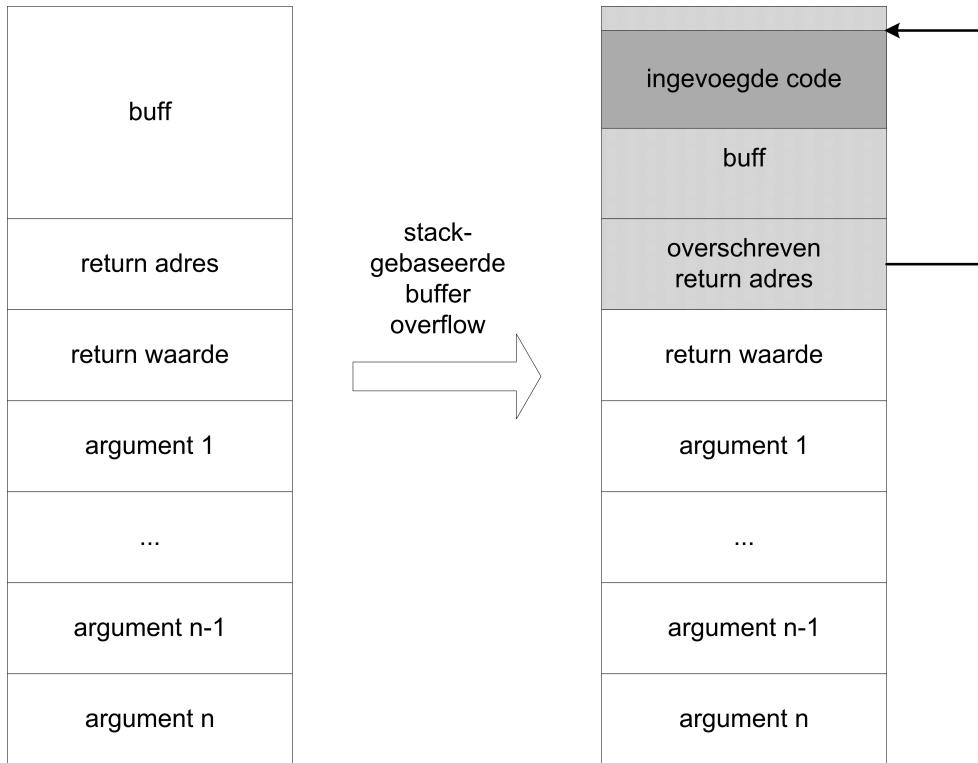
De hacker dient dus de lengte van de buffer te schatten (door trial en error bijvoorbeeld) en ook het adres op de stack te kennen van de buffer.

De techniek heeft als eigenschap dat de ingevoegde code kleiner moet zijn dan de grootte van de buffer. Daarom is er ook nog een tweede techniek waarbij aan de hand van een stack-gebaseerde buffer men het return adres overschrijft met het adres van een codefragment in een bibliotheekfunctie waarin ook een buffer overflow kan gerealiseerd worden.

Heap-gebaseerde buffer overflow

Een voorbeeld van een programma dat kwetsbaar is voor heap-gebaseerde overflow wordt hieronder getoond:

```
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    char* buff;
    if(argc <2){
        printf("Syntax: %s <input string>\n", argv[0]);
        exit (0);
    }
    buff = (char *)malloc(100*sizeof(char));
    strcpy(buff, argv[1]);
}
```



Figuur 7.3: Principe van stack-gebaseerde buffer overflow en uitvoering van ongewenste code.

Uitbuiten van heap-gebaseerde buffer overflow is gebaseerd op een ander principe dan stack-gebaseerde buffer overflow. Vermits de heap enkel programma data bevat, die at runtime gealloceerd wordt, is het minder evident om een ingevoegd stuk code te laten uitvoeren. De heap datastructuur is een dubbel gelinkte lijst, die alle gealloceerde stukken geheugen met elkaar linkt. De elementen van de dubbel gelinkte lijst bevatten de adressen van de vorige en volgende elementen en ook de grootte van het gereserveerde geheugen en het effectief gebruikte geheugen. Door een buffer te overschrijven kunnen valse headers ingevoegd worden, die zorgen voor het oproepen van geheugenbeheer functies (zoals een `free`-functie om geheugen vrij te geven en een leeg element uit de gelinkte lijst te verwijderen). Door te zorgen dat één van de pointers uit de gelinkte lijst het return adres van de `free`-functie bevat, kan dit return adres overschreven worden met het adres van de geïnserteerde code.

Door het uitvoeren van extra testen in de `free`-functie (bijvoorbeeld of de adressen wel naar geldig heap geheugen verwijzen en of de lijst nog wel mooi dubbel gelinkt is) kan dit gedrag voorkomen worden. Echter, het is zeer belangrijk dat de programmeur ervoor zorgt dat buffers overflow NIET kan optreden. De gevolgen van buffer overflow zijn immers altijd desastreus (onmiddellijk, als het programma erdoor crasht, of na een tijdje, als een hacker deze overflow op een creatieve manier misbruikt en vervolgens zijn code

wereldkundig maakt).

Vermijden van buffer overflows

De beste tactiek om buffer overflow te voorkomen bij gebruik van `strcpy` is om bij een kopieer-operatie de lengte van het te kopiëren blok te beperken tot de beschikbare ruimte. De functie `strncpy` bijvoorbeeld heeft volgende syntax:

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Deze functie kopiëert de eerste `n` karakters van `s2` naar `s1`. Indien het einde van de string `s2` (afsluitende '\0') tegengekomen wordt vooraleer de `n` karakters zijn gekopiëerd, dan wordt `s1` verder opgevuld met '\0' karakters tot wanneer er exact `n` karakters zijn geschreven naar `s1`.

Er wordt niet noodzakelijk een '\0' toegevoegd in `s1` (in tegenstelling tot `strcpy`), dus `s1` bevat enkel een geldige string als de lengte van `s2` kleiner is dan `n`.

Volgende aangepaste code zorgt ervoor dat het return adres niet meer kan overschreven worden:

```
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    char buff[100];
    if(argc <2){
        printf("Syntax: %s <input string>\n", argv[0]);
        exit (0);
    }
    strncpy(buff, argv[1], 100);
    return 0;
}
```

en idem voor de code voorbeeld met heap-gebaseerde buffer overflow.

In plaats van de grootte van een buffer hard te coderen, is gebruik van `sizeof` meer aangewezen:

```
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    char buff[100];
    if(argc <2){
        printf("Syntax: %s <input string>\n", argv[0]);
        exit (0);
    }
    strncpy(buff, argv[1], sizeof(buff));
    return 0;
}
```

Gebruik van `strlen`, zoals hieronder, is NIET correct. Waarom niet?

```
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    char buff[100];
    if(argc <2){
        printf("Syntax: %s <input string>\n", argv[0]);
        exit (0);
    }
    strncpy(buff, argv[1], strlen(buff)); /* foutief */
    return 0;
}
```

Op een gelijkaardige manier als `strcpy` kunnen de functies `strcat`, `sprintf` ook beveiligd worden tegen buffer overflows:

- `char* strncat(char *s1, const char *s2, size_t n);` concateneert de eerste `n` karakters van `s2` aan `s1`, er wordt steeds een '\0' aan het resultaat toegevoegd (net zoals de `strcat` functie). Uiteraard dient `s1` dient voldoende plaats te bevatten (`strlen(s1)+n+1`) voor de concatenatie. De strings `s1` en `s2` dienen onafhankelijk te zijn (i.e. niet naar elkaar verwijzen), anders is het resultaat ongedefinieerd.
- `int snprintf(char* s, size_t n, const char* ctrl);` zelfde gedrag als `sprintf`, maar ten hoogste `n` bytes (inclusief de terminerende '\0') worden gekopieerd naar `s`. De return waarde is het aantal bytes dat in `s` werd geschreven, zonder de terminerende '\0' mee te tellen.

Problemen met `strncpy` functie

Er zijn twee problemen met het gebruik van de `strncpy` functie:

1. Zoals beschreven hierboven is er niet noodzakelijk een terminerende '\0' in het resultaat aanwezig.
2. Indien `n` significant groter is dan de lengte van `s2` dienen heel veel '\0' karakters gekopieerd te worden en kan dit de performantie negatief beïnvloeden (i.e. het programma vertragen).

Omwille van deze redenen is het gebruik van de `snprintf` functie iets meer aanbevolen dan `strncpy`.

In Microsoft Visual C++ hebben de functies `strcpy`, `strncpy`, `snprintf` allen het label verouderd (Eng.: deprecated) gekregen. Volgende functie wordt in Microsoft Visual C++ naar voor geschoven in plaats van `strcpy` en `strncpy`: `strcpy_s` (s van secure), die bovenstaande twee problemen oplost.

Vermits andere platformen deze functie niet ondersteunen, zorgt het gebruik ervan voor

geen portabele code en dient het gebruik ervan vermeden te worden! Het beste is om bij veelvuldig gebruik van string manipulaties een aparte portabele library hiervoor te ontwerpen of een bestaande te gebruiken.

In het vervolg van de cursus en de practica zullen we steeds waar zinvol van de `strncpy` functie gebruikmaken (en steeds aandacht hebben voor de twee beschreven problemen).

Beveiligen van `scanf` en `sscanf`

Net zoals bij de copieer-functies, kan het gebruik van inlees-functies ook een buffer overflow tot gevolg hebben. Volgend codefragment illustreert dit:

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

    char buff[25];
    printf("please specify command:\n");
    scanf("%s", buff);
    ...
    return 0;
}
```

Deze code dient beschermd te worden door een limiet te zetten op de stringlengte:

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

    char buff[25];
    printf("please specify command:\n");
    scanf("%24s", buff);
    ...
    return 0;
}
```

namelijk 1 minder dan de buffergrootte (zodat '\0' ook kan opgeslagen worden).

Op een gelijkaardige manier dient men ook de `sscanf` functie te beschermen (door een limiet op de lengte van de individuele substrings te zetten).

In sommige applicaties zijn de formaat-strings dynamisch (i.e. worden bijgehouden in bestanden en dan ingelezen en gebruikt in de applicatie). Het hoeft geen betoog dat als een gebruiker toegang heeft tot deze formaat-strings hij de limieten kan aanpassen en het programma kwetsbaar maken voor de effecten van buffer overflows.

Voor de volledigheid: men dient ook voorzichtig te zijn als de gebruiker een string door geeft (via commandline, als programma-argument of via een bestand) en deze string wordt vervolgens door `printf` afgedrukt (`printf(input_string);`). De ingegeven string kan namelijk een formaat-string zijn (en dus conversie karakters bevatten): op die manier kan men de adressen op de stack te weten komen, via een pointer het return adres overschrijven en laten wijzen naar ingelezen code. Deze aanval wordt een *format string attack* genoemd en werkt dus zonder een buffer overflow! Het is dan aan de programmeurs om te controleren dat een input van een gebruiker geen formaat-strings bevat! De optie `printf("%s", input_string);` is dus veel veiliger dan `printf(input_string);`

7.6 Multidimensionele rijen

Een statische meerdimensionele rij is een rij van rijen, waarbij elementen NA elkaar opgeslagen worden in het geheugen (m.a.w. alle rijen aanéénslijtend na elkaar in het geheugen).

7.6.1 Declaratie

Declaratie gebeurt aan de hand van dubbele [] (twee-dimensionale rij) of drie-dubbele [] (drie-dimensionale rij). Het voorbeeld hieronder illustreert dit:

```
#define P 5
#define Q 2
#define R 8

int matrix_int[P][Q];
double matrix_double[P][Q][R];
```

7.6.2 Initialisatie

Initialisatie gebeurt aan de hand van geneste {}. Ontbrekende waarden worden op 0 geplaatst (net zoals bij ééndimensionale rijen).

7.6.3 Adressering

Aan de hand van de indices kan men het adres van een element bepalen (omdat alle elementen aanéénslijtend in het geheugen zijn opgeslagen). Volgende uitdrukkingen zijn dan ook equivalent voor statische twee-dimensionale rijen:

```
matrix[i][j]
*(matrix[i]+j)
*((*(matrix+i))+j)
```

Vraag: op het eerste zicht is de volgende uitdrukking ook equivalent met de drie voorgaande uitdrukkingen:

```
*(matrix+Q*i+j)
```

dit is echter niet het geval! Verklaar. Hoe kan je deze uitdrukking lichtjes aanpassen zodat ze wel een resultaat geeft dat equivalent is met het resultaat van de drie voorgaande uitdrukkingen?

Voor drie-dimensionale rijen zijn volgende uitdrukkingen equivalent:

```
naam[i][j][k]
*(naam[i][j]+k)
```

Om dezelfde reden als bij twee-dimensionale rijen is de volgende uitdrukking echter niet equivalent:

```
*(naam+(Q*R)*i+R*j+k)
```

Vraag: hoe kan je deze uitdrukking lichtjes aanpassen zodat ze wel het gewenste resultaat geeft?

Bij statische rijen dient men op voorhand de dimensies te kennen. Indien men deze pas tijdens de uitvoering van een programma kent, dient men dynamisch gealloceerde rijen te gebruiken! Deze komen hieronder aan bod.

7.6.4 Meerdimensionale rijen: wijzerrijken

Bij dynamisch gealloceerde rijen, werkt men steeds met een ééndimensionale array van wijzers, waarbij elke wijzervariabele op zijn beurt naar een array verwijst.

Het voorbeeld hieronder toont een array van strings, die geïnitialiseerd wordt:

```
void signal_runtime_error (int num){
    static char *error[] =
    { "Ethernet chip not initialised. \n",
      "Device not ready. \n",
      "Input Buffer overflow detected. \n",
      "Transmission problem .\n"
    };
    printf("SYSTEM> %s.\n",error[num]);
}
```

7.6.5 Meerdimensionale rijen: programma-argumenten

Het doorgeven van commandolijn-argumenten aan programma's gebeurt ook aan de hand van een array `argv` van karakter strings. De waarde van het argument `argc` geeft weer hoeveel karakter strings er aanwezig zijn in `argv`. `argv[0]` is meestal de naam van het programma, `argv[1]` is het eerste argument, `argv[2]` is het tweede, ..., `argv[argc - 1]` is het laatste, en `argv[argc]` is een NULL pointer.

Volgend programma geeft de commandolijn-argumenten weer via `printf`:

```
/* argumenten.c */
#include <stdio.h>

int main(int argc,char* argv[ ]) {
    int i=0;
    printf("Programma-argumenten : \n");
    for(i=0;i<argc;i++)
        printf("%s\n",argv[i]);
    return 0;
}
```

7.6.6 Meerdimensionale rijen: dynamische rijen

Zoals gezien in sectie 7.4 kunnen ééndimensionale rijen als volgt aangemaakt worden:

```
#define P 5
...
int* a;
...
a=(int*)calloc(P,sizeof(int));
...
```

en vernietiging gebeurt door:

```
...
free(a);
...
```

Bij tweedimensionale rijen, werkt men steeds met een ééndimensionale array van wijzers, waarbij elke wijzervariabele op zijn beurt wijst naar een ééndimensionale array van waarden. De allocatie hiervan gebeurt dan als volgt:

```
#define P 5
#define Q 2
...
int** a;
int i;
...
a=(int**)calloc(P,sizeof(int*));
for(i=0;i<P;i++)
    a[i]=(int*)calloc(Q,sizeof(int));
...
```

Eerst wordt de array van wijzervariabelen aangemaakt en vervolgens de individuele rijen zelf. Figuur 7.4 toont schematisch de structuur van de aangemaakte tweedimensionale array. Vernietiging van deze tweedimensionale rij gebeurt als volgt:

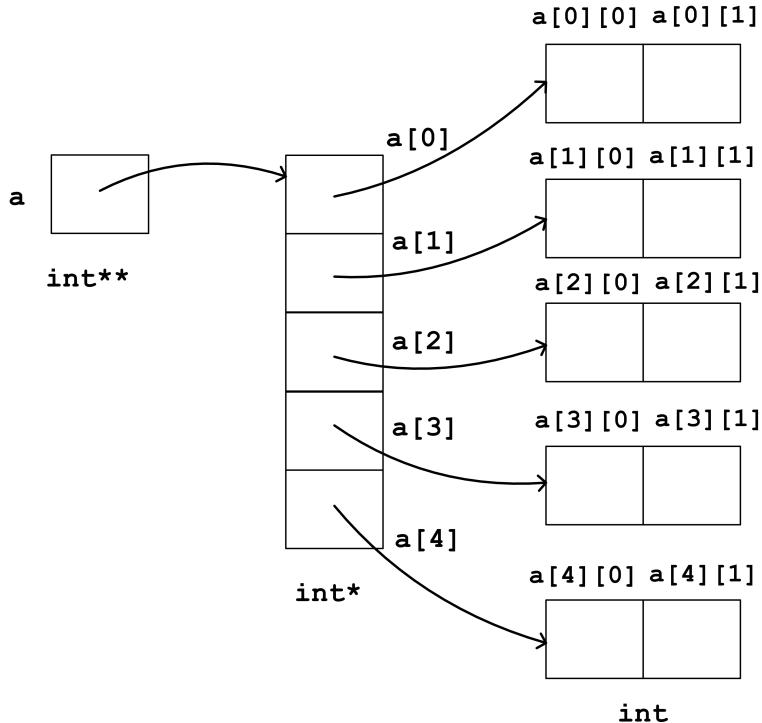
```
...
for(i=0;i<P;i++)
    free(a[i]);
free(a);
...
```

Met andere woorden: vernietiging van de rijen gebeurt in omgekeerde volgorde van de allocatie.

7.7 Functie-argumenten

7.7.1 Principe

Functies hebben ook een adres dat aan andere functies kan doorgegeven worden als argument of door een functie als return-waarde kan teruggegeven worden. De variabele



Figuur 7.4: Schematische voorstelling in het geheugen van een tweedimensionale array, via dynamisch geheugenbeheer aangemaakt.

die dit adres bevat, wordt een *functiewijzer* (Eng.: function pointer) genoemd: deze variabele bevat het adres van de eerste instructie van de functie.

De klassieke declaratie van een functie is als volgt:

```
type naam(proto_lijst)
```

De declaratie van een functiewijzer gebeurt als volgt:

```
type (*naam)(proto_lijst)
```

naam is dan de wijzer naar de functie, ***naam** is de functie zelf en een functieoproep via de functiewijzer vindt als volgt plaats:

```
(*naam)(actuele_lijst)
```

Dereferentie gebeurt automatisch in de functie zelf, dus volgende twee uitdrukkingen zijn equivalent:

```
*naam(actuele_lijst)
```

```
naam(actuele_lijst)
```

7.7.2 Voorbeeld 1: automatisch testen van functies

Hierbij wordt het gebruik van een functiewijzer `f_ptr` geïllustreerd voor automatisch testen van functies:

```
void signal_runtime_error(int);
void signal_memory_error(int);
...
int i=0;
void (*f_ptr)(int);

f_ptr= signal_runtime_error ;
for(i=0;i<4;i++)
    f_ptr(i);

f_ptr= signal_memory_error ;
for(i=0;i<4;i++)
    f_ptr(i);
```

7.7.3 Voorbeeld 2: tabulatie van functiewaarden

In dit voorbeeld wordt een functie getoond, die een andere functie als argument neemt. Er wordt een functiewijzer als argument doorgegeven. Via dit argument wordt de doorgegeven functie dan opgeroepen:

```
/* functiewijzer.c */
#include <stdio.h>
#include <math.h>

double kwadraat(double);
void tabuleer(double f(double),double,double,double);

int main(void) {
    printf("Tabel van sin(x)\n");
    tabuleer(sin,0.0,1.57,0.2);
    printf("Tabel van x*x\n");
    tabuleer(kwadraat,0.0,1.0,0.2);
    return 0;
}

void tabuleer(double f(double),double s,double e,double stap) {
    double x;
    printf("-----\n");
    for(x=s;x<e;x+=stap)
        printf("%f -> %f\n",x,f(x));
```

```

    printf("-----\n");
}

double kwadraat(double x) {
    return x*x;
}

```

7.7.4 Functie-argumenten: rijen van functiewijzers

Indien men rijen van functiewijzers wil aanmaken dient men via `typedef` eerst een apart type voor de functiewijzer aan te maken. Dit wordt hieronder getoond voor een rij van 10 functiewijzers, waarbij elk element het adres bevat van een functie met twee `int`-argumenten en een `int` als returnwaarde:

```

typedef int (*discretef)(int,int);
discretef f[10];

```

Hieronder wordt het voorbeeld van daarnet (sectie 7.7.3) getoond, waarbij nu een rij van 2 functiewijzers gebruikt wordt in de `main`-functie:

```

/* functierij.c */
#include <stdio.h>
#include <math.h>

double kwadraat(double);
void tabuleer(double f(double),double,double,double);
typedef double (*realf)(double);

int main(void) {
    realf f[2]={sin,kwadraat};
    int i;
    for(i=0;i<2;i++) tabuleer(f[i],0.0,1.0,0.2);
    return 0;
}

void tabuleer(double f(double),double s,double e,double stap) {
    double x;
    printf("-----\n");
    for(x=s;x<e;x+=stap)
        printf("%f -> %f\n",x,f(x));
    printf("-----\n");
}

double kwadraat(double x) {
    return x*x;
}

```

7.8 Struct's in combinatie met wijzers

Zoals in hoofdstuk 4 aangebracht werd, kan men op de volgende manier een struct aanmaken:

```
typedef struct{
    char naam[30];
    short leeftijd;
    char code;
} persoon1;
...
persoon1 p1,p2;
...
```

De struct heeft als typenaam `persoon1` (verder in deze sectie komt ook een struct `persoon2` aan bod, die een variant is van `persoon1`). In dit voorbeeld zijn `p1.naam` en `p2.naam` beide STATISCHE rijen, waarbij geheugen wordt gealloceerd tijdens creatie van `p1` en `p2`.

7.8.1 Struct's en pointers

Een pointer naar een variabele van het type `struct` verdient een speciale vermelding. Toegang tot een member van deze struct is voor de hand liggend: stel dat de naam van de struct-variabele `s` is en `p = &s`, dan levert `(*p).a` bijvoorbeeld de waarde van veld `a`. Om de haakjes te vermijden bestaat ook de volgende veelgebruikte notatie: `p->a`.

Volgende code illustreert dit:

```
typedef struct{
    char naam[30];
    short leeftijd;
    char code;
} persoon1;
...
persoon1 p;
persoon1* w;
w=&p;
...
```

Volgende uitdrukkingen

```
(*w).naam
(*w).leeftijd
(*w).code
(*w).naam[1]
```

kunnen ook verkort als volgt genoteerd worden:

```
w->naam
w->leeftijd
w->code
(w->naam)[1]
```

Met andere woorden, de volgende uitdrukking:

```
(*struct_wijzer).veldnaam
```

is steeds equivalent met:

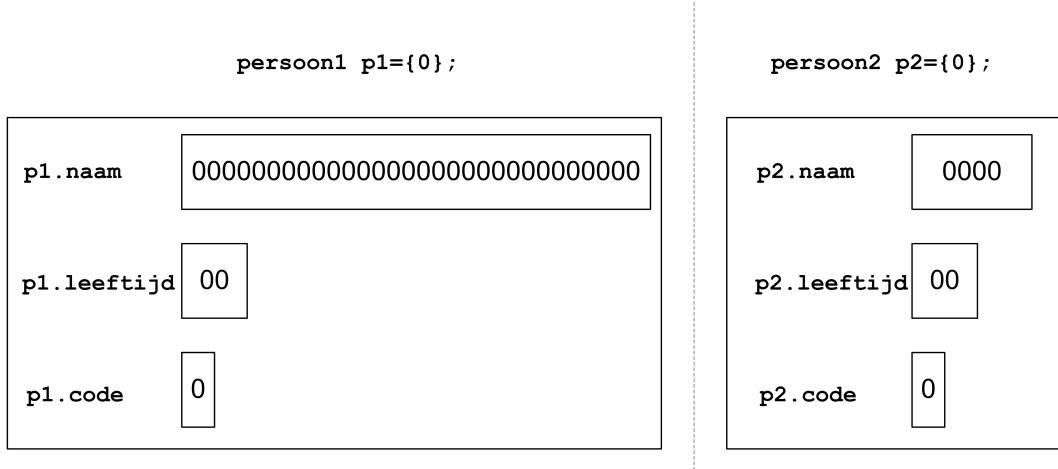
```
struct_wijzer->veldnaam
```

7.8.2 Struct's: wijzervelden

Beschouwen we nu een struct van type `persoon2`, deze bevat een wijzerveld (`char* naam` in tegenstelling tot de statische rij `char naam[30]` bij `persoon1`):

```
typedef struct{
    char* naam;
    short leeftijd;
    char code;
} persoon2;
```

Wanneer we `sizeof(persoon2)` berekenen is deze aanzienlijk kleiner dan `sizeof(persoon1)`: er wordt GEEN geheugen voor de inhoud van het veld `naam` gereserveerd! Figuur 7.5 toont schematisch de voorstelling van een variabele van type `persoon1` en van type `persoon2`.



Figuur 7.5: Schematische voorstelling in het geheugen van een variabele van type `persoon1` en van type `persoon2`.

`p2.naam` is een dynamisch te alloceren rij. Het type `persoon2` vergt functies voor alloctatie en vernietiging. Een voorbeeld van dergelijke functies `init_persoon2` en `free_persoon2` wordt hieronder weergegeven:

```

persoon1 p1={0};
persoon2 p2={0};

#define NAAML 30

typedef struct{
    char *naam;
    short leeftijd;
    char code;
} persoon2;

void init_persoon2(persoon2* );
void free_persoon2(persoon2);

int main(void) {
    persoon2 p2={0};
    init_persoon2(&p2);
    free_persoon2(p2);
    return 0;
}

void init_persoon2(persoon2* p) {
    if(p->naam != NULL) free_persoon2(*p);
    p->naam=(char*)calloc(NAAML,sizeof(char));
    if (p->naam != NULL) p->naam[0]='\0';
}

void free_persoon2(persoon2 p) {
    if (p.naam !=NULL) free(p.naam);
}

```

7.8.3 Struct's: soorten kopieën

Beschouwen we volgende code, waarbij twee struct-variabelen van type **persoon1** aan elkaar toegekend worden:

```

persoon1 a={"Jan Jansen",30,'A'};
persoon1 b={0};
b=a;

```

Hierbij wordt een DIEPE KOPIE (Eng.: deep copy) uitgevoerd: de beide structs zijn mooi gescheiden in het geheugen.

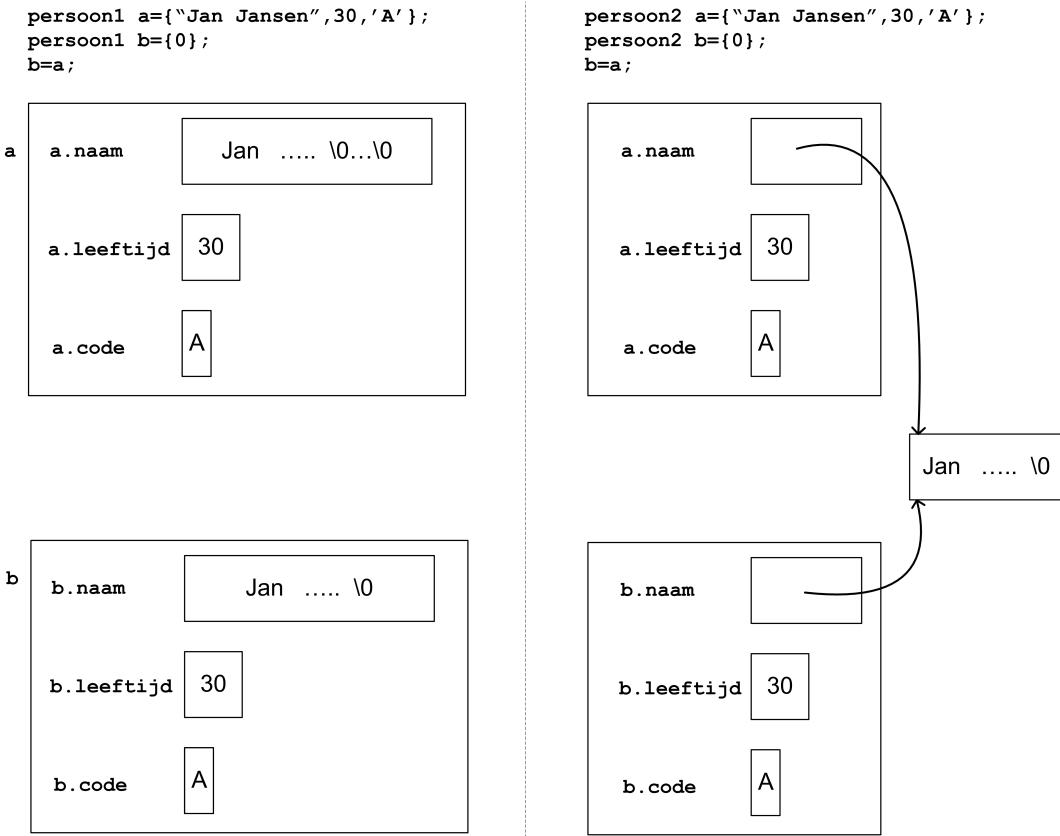
Wanneer twee struct-variabelen van type **persoon2** aan elkaar toegekend worden:

```

persoon2 a={"Jan Jansen",30,'A'};
persoon2 b={0};
b=a;

```

vindt er een ONDIEPE KOPIE (Eng.: shallow copy) plaats: de beide structs bevatten een veld, waarbij beiden wijzen naar hetzelfde geheugen. Het zijn dus geen onafhankelijke kopiëen. Figuur 7.6 toont schematisch het verschil tussen een kopie van twee variabelen van type `persoon1` en van type `persoon2`.



Figuur 7.6: Schematische voorstelling in het geheugen van een kopie van twee variabelen van type `persoon1` (diepe kopie) en van type `persoon2` (ondiepe kopie).

Volgende code toont het effect van een ondiepe kopie:

```
int main(void) {
    persoon2 a={0}; persoon2 b={0};
    init_persoon2(&a,"Jan Jansen",30,'A');
    init_persoon2(&b,"Piet Verlinden",35,'B');
    print_persoon2(a);print_persoon2(b);
    b=a;b.code='N';
    wijzig_naam(&b,"Mario Puzo");
    print_persoon2(a);print_persoon2(b);
    free_persoon2(a);free_persoon2(b);
    return 0;
}
...
```

```

void print_persoon2(persoon2 p) {
    printf("Naam = %s\n",p.naam);
    printf("Leeftijd = %d\n",p.leeftijd);
    printf("Code = %c\n",p.code);
}

void wijzig_naam(persoon2* p,const char* n) {
    p->naam=(char*)realloc(p->naam, (strlen(n)+1)*sizeof(char));
    strcpy(p->naam,n);
}

```

Na de ondiepe kopie wordt één van de beide struct's aangepast: de andere struct verandert mee!

7.8.4 Oefening: diepe kopie

Schrijf een functie die een diepe kopie maakt van een variabele van het type `persoon2`. De oplossing wordt hieronder voor de volledigheid weergegeven:

```

void deep_copy(persoon2* toPersoon, const persoon2* fromPersoon) {
    int naamlengte=0;
    naamlengte = strlen(fromPersoon->name);

    toPersoon->name=(char*)calloc(naamlengte+1,sizeof(char));
    toPersoon->name=strcpy(toPersoon->name, fromPersoon->name);

    toPersoon->leeftijd=fromPersoon->leeftijd;
    toPersoon->code=fromPersoon->code;
}

```

7.8.5 Struct's en functies

Struct's kunnen als functie-argument doorgegeven worden of als returnwaarde van een functie teruggegeven worden. Deze beide opties komen hieronder aan bod:

Struct als argument

Een struct wordt hierbij behandeld als elk ander type (*pass-by-value*): er wordt een ONDIEPE kopie genomen (Eng.: shallow copy). Zoals reeds aan bod kwam, worden dynamisch gealloceerde variabelen niet automatisch in een nieuwe geheugenruimte gekopieerd!

Struct als return type

Een struct wordt behandeld als elk ander type (m.a.w. rijen van struct's zijn niet als resultaat toegelaten, wel struct*)

Voorbeeld

In dit voorbeeld worden twee **wissel**-functies geïllustreerd:

1. de ene functie neemt een array van twee elementen als argument
2. de andere functie neemt een struct als argument, deze struct bevat een array van twee elementen als veld

Vraag: Welke output geeft dit programma? Verklaar!

```
#include <stdio.h>
#define PRINT2(x)  (printf("el1=%d, el2=%d\n",x[0],x[1]))

void wissel(int[ ]);
void wissel_rij2(rij2);

typedef struct {
int r[2];
} rij2;

int main(void) {
int a[ ]={1,2};
rij2 b={{1,2}};
PRINT2(a);
wissel(a); PRINT2(a);

PRINT2(b.r);
wissel_rij2(b); PRINT2(b.r);
wissel(b.r); PRINT2(b.r);
return 0;
}

void wissel(int a[ ]) {
int t;
t=a[0];
a[0]=a[1];
a[1]=t;
}

void wissel_rij2(rij2 s) {
int t;
t=s.r[0];
s.r[0]=s.r[1];
s.r[1]=t;
}
```

7.8.6 Struct's: geneste structuren

Een struct kan als veld een andere struct bevatten: dit worden geneste struct's genoemd. Het voorbeeld hieronder toont een struct **naam**, die als veld in de struct **persoon** gebruikt

wordt:

```
typedef struct {
    char* voor;
    char* fam;
} naam;

typedef struct {
    naam n;
    int leeftijd;
    char code;
} persoon;

persoon p;
persoon* q;
naam* r;

...
q=&p;
r=&(p.n);
printf("Voornaam = %s",p.n.voor);
```

In dit voorbeeld is

`p.n.voor`

equivalent met volgende uitdrukkingen:

```
(*r).voor
r->voor
(*q).n.voor
(q->n).voor
```


Hoofdstuk 8

Andere verschillen met Java

In dit hoofdstuk wordt verder ingegaan op de verschillen met Java.

8.1 const in plaats van final

Java gebruikt het sleutelwoord `final` om aan te geven dat variabelen slechts éénmaal een waarde kunnen krijgen (meestal bij declaratie). C gebruikt het sleutelwoord `const` bij de declaratie van een constante, die bij declaratie moet geïnitialiseerd worden. Een constante heeft wel een adres en een grootte, die kunnen opgevraagd worden.

Constanten zijn zeer nuttig bij de declaratie van functies die pointers of array's als argumenten nemen, maar die de pointer-waarde of de inhoud van de array's niet (mogen) aanpassen.

Een constante pointer kan ook gedefinieerd worden als de pointer niet naar elders mag verwijzen dan naar het adres, dat hem bij initialisatie toegekend werd.

8.2 inline functies

In C kunnen functies `inline` gedeclareerd worden. Dit wordt vooral gebruikt voor functies die zeer snel moeten uitgevoerd worden. De compiler plaatst dan een kopie van de functie-code op de plaats in het bronbestand waar de functie wordt opgeroepen in plaats van de functie-oproep. Op deze manier wordt de overhead vermeden om de functie op te roepen. Inline functies staan dikwijls in de headerbestanden in plaats van hun declaraties.

Hierbij een voorbeeld van een inline functie:

```
inline int max (int a, int b) {
    if (a > b) return a;
    else return b;
}
```

8.3 Afwezigheid van exceptions

Java ondersteunt exceptions ter afhandeling van enerzijds applicatie-gedefinieerde fouten en anderzijds meer ernstige systeem- of geheugentoegangsfouten (zoals toegang buiten de grenzen van een array).

In C worden applicatie-gedefinieerde foutcondities meestal uitgedrukt door zorgvuldige definitie van de betekenis van de return-waarden van functies.

Meer ernstige fouten, zoals een poging van toegang tot geheugen dat niet gealloceerd is, kan ongemerkt plaatsvinden met ongedefinieerd gedrag tot gevolg. Schrijftoegang tot zulk geheugen kan corruptie van kritieke data veroorzaken, die slechts later zichtbaar wordt.

Dergelijke fouten zijn dus moeilijk op te sporen omdat men niet noodzakelijk een foutmelding krijgt van de compiler of bij het uitvoeren van het programma, en het programma soms vastloopt en soms niet.

In Java en C++ wordt gewerkt met een **try catch** blok, waarbij in het **catch** gedeelte mogelijke opgeworpen exceptions in het blok, opgevangen worden en kunnen afgehandeld worden (i.e. code kan voorzien worden om op de gepaste manier te reageren op het voorkomen van de exception, deze code wordt ook *exception handler* code genoemd).

```
try{
    ...
} catch(Exception e)
{
    ...
}
```

8.4 Type aliasing

In tegenstelling tot Java kent C het sleutelwoord **typedef**. Dit laat toe om nieuwe namen of aliases voor bestaande types te definiëren.

Een voorbeeld hiervan werd reeds in hoofdstuk 4 bij **typedef** van struct's gegeven. Aliases worden altijd ter verkorting of verduidelijking gebruikt. Ook bij systeemtypes worden ze gebruikt, bijvoorbeeld:

```
typedef unsigned char byte; /* bijv. 1 byte */
typedef unsigned short word; /* bijv. 2 bytes */
typedef unsigned long dword; /* bijv. 4 bytes */
```

8.5 main() functie

In een Java applicatie begint de uitvoering in de statische methode **main** van een bepaalde klasse:

```
public static void main(String[] args){
    //uit te voeren code
}
```

`args` is een array van `String` objecten, waarbij elke string een opdrachtlijnargument is.

In C begint de uitvoering ook in de functie met naam `main`, maar deze heeft volgende signatuur:

```
int main(int argc, char **argv){
    /* uit te voeren code */
}
```

Het argument `argv` geeft een `array` van karakter strings weer, die het commando vormen om het programma uit te voeren. De waarde van het argument `argc` geeft weer hoeveel karakter strings er aanwezig zijn in `argv`. `argv[0]` is meestal de naam van het programma, `argv[1]` is het eerste argument, `argv[2]` is het tweede, ..., `argv[argc - 1]` is het laatste, en `argv[argc]` is een `NULL` pointer.

Bemerkt in C dat het return type en de vermelding van de parameters optioneel is, volgende versies zijn ook toegelaten:

```
void main(int argc, char **argv){
    /* uit te voeren code */
}

void main(void){
    /* uit te voeren code */
}

void main(){
    /* uit te voeren code */
}
```

8.6 Standaard bibliotheek

Java beschikt over een rijke verzameling van klassen om de ontwikkelaar ondersteuning te bieden voor I/O, netwerkoperaties, GUI's, etc. De programmeertaal C beschikt over een bibliotheek van functies, types en macro's om de ontwikkelaar te ondersteunen. Deze functies, types en macro's vormen de C Standaard Bibliotheek (Eng.: Standard Library). Deze bibliotheek is beperkt gehouden om de portabiliteit zo hoog mogelijk te houden (er is bijvoorbeeld geen ondersteuning voor GUI's). Bibliotheeken voor GUI's zijn meestal platformspecifiek.

In de tabel 8.1 wordt een overzicht gegeven van de headerbestanden van de C standaard bibliotheek.

<code><stddef.h></code>	Enkele essentiële macro's en additionele type declaraties
<code><stdlib.h></code>	Toegang tot de omgeving; dynamisch geheugenbeheer; verschillende hulp macro's en functies (Eng.: utilities)
<code><stdio.h></code>	Invoer en uitvoer
<code><string.h></code>	String-bewerkingen
<code><ctype.h></code>	Klassificatie van karakters (upper/lower case, alphabetic/numeric etc)
<code><limits.h></code>	Implementatie-gedefineerde limieten voor gehele types
<code><float.h></code>	Implementatie-gedefineerde limieten voor floating-point types
<code><math.h></code>	Wiskundige functies
<code><assert.h></code>	Diagnostische macro's en functies (Eng.: utilities)
<code><errno.h></code>	Error identificatie
<code><locale.h></code>	Regionale/nationale variaties van karakter sets, tijdsformaten, etc
<code><stdarg.h></code>	Ondersteuning voor functies met een variabel aantal argumenten
<code><time.h></code>	Voorstelling van tijd, en toegang tot de systeem-clock
<code><signal.h></code>	Behandeling van exceptionele run-time gebeurtenissen (Eng.: events)
<code><setjmp.h></code>	Herstelling van de uitvoering naar een vorige toestand

Tabel 8.1: Inhoud van de headerbestanden uit de standaard bibliotheek

8.7 Bitvelden en -operatoren

In tegenstelling tot Java, biedt C de mogelijkheid om de individuele bits in een byte te bewerken. Een aantal mogelijke toepassingen hiervan zijn:

1. het laat toe om efficiënt met geheugen om te gaan, bijv. de 8 bits van een byte kunnen elk een booleaanse variabele voorstellen,
2. bij communicatie met randapparatuur heeft iedere uitgewisselde bit een betekenis, *device drivers* dienen deze informatie dus te kunnen lezen en te kunnen invullen door de individuele bits te bewerken,
3. bij het schrijven van geavanceerde encryptie-routines dient men toegang te hebben tot individuele bits of reeksen van bits.

Een voorbeeld waarbij men alle bits zo optimaal mogelijk benut, is de declaratie van de struct `dag`:

```
typedef struct dag {
    unsigned maand:4;
    unsigned dag:5;
    unsigned jaar:12;
} dag;
```

Hierbij wordt een dag aan de hand van 4 bytes (één `int` op een 32-bits machine) voorgesteld, en de getallen na de dubbele punt (`:`) geven het volgende aan:

- er worden 4 bits voorzien om een maand voor te stellen (4 bits laat 2^4 of 16 mogelijke waarden toe, dus voldoende om 12 maanden voor te stellen),
- de volgende 5 bits worden gebruikt om een dag voor te stellen (5 bits laat 2^5 of 32 mogelijke waarden toe, dus voldoende om maximaal 31 dagen per maand voor te stellen),
- de volgende 12 bits dienen om het jaartal voor te stellen (12 bits laat 2^{12} of 4096 mogelijke waarden toe, dus voldoende ver in het verleden en de toekomst).

De totale grootte van een dergelijke struct wordt steeds afgerond naar de woordbreedte van de machine. Met andere woorden: anonieme bitvelden zorgen voor opvulling (dit wordt *padding* genoemd).

De syntax voor een bitveld van een struct (in BNF notatie) luidt als volgt:

```
bit_veld ::= {int | unsigned}_1 {naam}_{opt} : constante_gehele_uitdr
```

Indien bitvelden zonder teken gebruikt worden (a.d.h.v. `unsigned`) is de betekenis van de bits ondubbelzinnig. In geval bitvelden met teken gebruikt worden, is hun representatie machine-afhankelijk (afhankelijk bijvoorbeeld of een tekenbit of 2-complement voorstelling gebruikt wordt), hetgeen de portabiliteit negatief kan beïnvloeden. Daarom wordt veel `unsigned` gebruikt voor de declaratie van bitvelden.

Volgende beperkingen op bitvelden kunnen onderscheiden worden (in vergelijking met gewone types):

- rijen (Eng.: arrays) van bitvelden zijn niet mogelijk,
- adresbepaling via & op een bitveld is niet mogelijk.

8.8 Opsommingen

In C kan via het `enum` sleutelwoord een opsomming van symbolische constanten gedeclareerd worden. Symbolische constanten dienen in Java gedeclareerd te worden aan de hand van een `public static final` declaratie. Het sleutelwoord `enum` bestond lange tijd niet in Java, maar is sinds Java 1.5 toch ingevoerd.

Het doel van `enum` in C is om gehele types met een beperkt bereik zinvolle namen te geven en om de code duidelijker te maken. Volgend voorbeeld toont twee opsommingen:

```
enum seizoen {lente, zomer, herfst, winter};
enum maand {januari, februari, maart, april, mei, juni, juli,
            augustus, september, oktober, november, december};
```

Een opsomming wordt intern steeds als geheel type gestockeerd en mag overal in uitdrukkingen gebruikt worden waar een geheel type toegelaten is. De syntax (in BNF

notatie) luidt als volgt:

```
enum tag_naam{,naam{=constante_uitdr}opt}1+;
```

De toekenning van een constante_uitdrukking (steeds geheel) in een opsomming zorgt ervoor dat de naam dan steeds correspondeert met de opgegeven waarde, en de volgende elementen in de opsomming corresponderen met de opgegeven waarde + 1, opgegeven waarde + 2, etc. (tenzij deze ook een hogere waarde toegekend krijgen).

Een afkorting via `typedef` is uiteraard steeds mogelijk, zoals geïllustreerd in volgend voorbeeld:

```
typedef enum dag_{ma=1,di,wo,don,vr,za,zo} dag;

int main(void) {
    dag i;
    for(i=ma;i<=zo;i++)
        printf("i=%d\n",i);
}
```

Indien men de elementen van een opsomming als string wil afdrukken, dient men een conversie-functie te schrijven. Dit wordt hieronder geïllustreerd voor de opsomming van maanden:

```
#include <stdio.h>
enum maand {JAN=0, FEB, MAR, APR, MAY, JUN,
            JUL, AUG, SEP, OKT, NOV, DEC };
typedef enum maand maandType;
char* maandString(maandType);

int main(){
    maandType nieuweMaand;
    for (nieuweMaand = JAN; ; ){
        if ((int)nieuweMaand > (int)DEC) break;
        printf("%s \n", maandString(nieuweMaand));
        nieuweMaand = maandType ((int) nieuweMaand + 1);
    }
    return 0;
}

char* maandString(maandType dezeMaand){
    static char *maandString [] = {
        "January", "February", "March", "April",
        "May", "June", "July", "August",
        "September", "October", "November", "December"
    };
    return (maandString[dezeMaand]);
}
```

Hoofdstuk 9

Abstracte Datatypes

9.1 Definitie

Een *datastructuur* zorgt voor de organisatie van gegevens (in gelinkte lijsten, boomstructuren, tabellen, hashtabellen, etc.).

Een *datatype* is een datastructuur waarvan ook de typische operaties in rekening gebracht worden, bijv. zoeken, bijvoegen, verwijderen van een element, etc.

Een *abstract datatype* is een datatype waar onderscheid gemaakt wordt tussen de definities van de data en de operaties en hun implementatie. Met andere woorden, het datatype kan gebruikt worden, zonder kennis van de implementatie, enkel de interface dient gekend te zijn. Dit wordt verder toegelicht in volgende sectie en geïllustreerd aan de hand van een uitgebreid code-voorbeeld in sectie 9.3. Vaak wordt ADT gebruikt als afkorting van abstract datatype.

9.2 Implementatie in C

Een toepassing in C, die gebruik maakt van een ADT bestaat minstens uit 3 bestanden:

1. een interface, die de datatypes beschrijft samen met de functies/operaties die toegelaten zijn op het ADT. Deze wordt opgeslagen in een headerbestand, bijv. `ADT.h`.
2. een implementatie van de functies en operaties, die in het headerbestand gedeclareerd zijn. Deze implementatie wordt opgeslagen in een `.c` bestand, bijv. `ADT.c`.
3. een toepassings- of client programma, dat gebruik maakt van het ADT. Dit programma includeert het headerbestand `ADT.h` en gebruikt enkel de functies van de ADT om abstracte operaties uit te voeren. Deze implementatie wordt bijvoorbeeld opgeslagen in het bestand `client.c`.

De code in het bestand `ADT.c` kan apart (i.e. zonder het toepassings- of client programma) gecompileerd worden en levert dan het objectbestand `ADT.o` op. Voor de compilatie van de client code is enkel het bestand `client.c` en het headerbestand `ADT.h`

nodig, hetgeen dan `client.o` oplevert. Om een uitvoerbaar bestand te bekomen, dient uiteraard `ADT.o` en eventueel code uit de standaardbibliotheek (`libc.a`) gelinkt te worden met `client.o`.

In plaats van de code van het ADT via `ADT.o` ter beschikking te stellen, kan het ook in een bibliotheek (Eng.: library) opgenomen worden. Meestal gebeurt dit wanneer een significant aantal ADT's ontwikkeld zijn en nuttig zijn om te groeperen in een bibliotheek.

9.3 Voorbeeld: stapel (Eng.:stack)

9.3.1 Situering

Een stapel is een LIFO (Eng.: Last In First Out) datastructuur, waarin gegevens opgeslagen worden. Een nieuw element wordt telkens achteraan toegevoegd en bij het afhalen van een element wordt telkens het laatst toegevoegde element genomen en van de stapel verwijderd.

In dit voorbeeld zullen we een stapel van gehele getallen (datatype `int`) als abstract datatype implementeren.

9.3.2 Bewerkingen

Zoals hierboven gesitueerd, zijn volgende twee operaties op de stapel zinvol:

1. `push`: element bijplaatsen op de stapel, indien vol dient een foutcode teruggegeven te worden.
2. `pop`: element afhalen van de stapel, indien de stapel leeg is, dient eveneens een foutcode teruggegeven te worden.

9.3.3 Interface

Als datastructuur wordt in dit voorbeeld voor een struct gekozen, die een array van vaste lengte bevat, samen met een geheel getal `tos` (Eng.: top of stack) dat aanduidt tot waar de stapel gevuld is.

```
#define SIZE 25

typedef struct {
    int d[SIZE];
    int tos;
} stack;
```

Voor de `push` functie wordt volgende declaratie gebruikt:

```
int push(stack*,int);
```

Deze functie geeft +1 terug indien gelukt, en geeft 0 terug indien niet gelukt.

Voor de `pop` functie wordt volgende declaratie gebruikt:

```
int pop(stack*, int*);
```

Deze functie geeft +1 terug indien gelukt, en geeft 0 terug indien niet gelukt. Het tweede argument bevat na oproepen van de functie het afgehaalde element (indien de return-waarde aangeeft dat de operatie gelukt is).

Het correct initialiseren van het `tos` veld is zeer belangrijk: om het ADT abstract te houden voor de gebruikers is het belangrijk dat de gebruikers zelf dit `tos` veld niet hoeven te initialiseren. Hiervoor wordt de functie `reset` voorzien, die zorgt voor het ledigen en initialiseren van de stapel. Tenslotte zijn er ook de functies `full` en `empty`, die toelaten om te controlen of de stapel respectievelijk vol of leeg is.

Het bestand `stack.h` bevat dan volgende code:

```
#define SIZE 25
#define EMPTY -1
typedef struct {
    int d[SIZE];
    int tos;
} stack;

int push(stack*, int);
int pop(stack*, int*);
void reset(stack*);
int full(stack*);
int empty(stack*);
```

Opgave: probeer zelf de implementatie van de 5 functies te schrijven. Ter controle wordt de oplossing hieronder weergegeven.

9.3.4 Implementatie

In het bestand `stack.c` komt dan de implementatie van alle functies:

```
#include "stack.h"

int full(stack* s){
    if (s->tos==SIZE-1) return 1;
    else return 0;
}

int empty(stack* s){
    if (s->tos==EMPTY) return 1;
    else return 0;
}
```

```

void reset(stack* s){
    s->tos=EMPTY;
}

int push(stack* s,int i) {
    int r;
    if(r!=full(s)) {
        s->tos++;
        s->d[s->tos]=i;
    }
    return r;
}

int pop(stack* s,int* i) {
    int r;
    if(r!=empty(s)) {
        *i=s->d[s->tos];
        s->tos--;
    }
    return r;
}

```

9.3.5 Gebruik

In een client bestand kan men dan de stapel ADT gebruiken. Dit wordt geïllustreerd aan de hand van onderstaande code, waarbij gegevens in een stapel opgeslagen worden en vervolgens in omgekeerde volgorde van toevoegen de elementen weergegeven worden:

```

/* reverse.c */
#include <stdio.h>
#include "stack.h"

int main(void) {
    int g;
    stack s;
    reset(&s);

    do {
        printf("Geef een positief getal :");
        scanf("%d",&g);
        if(g>0) push(&s,g);
    } while(g>0);

    printf("Afbeelden van de getallen : ");

    while(pop(&s,&g))
        printf("%d\t",g);
}

```

9.3.6 Uitbreiding

In bovenstaand voorbeeld is de grootte van de stapel statisch (i.e. éénmaal gecompileerd kan deze niet meer aangepast worden). Opgave: breid het voorbeeld uit, zodat via dynamisch geheugenbeheer steeds juist voldoende geheugen beschikbaar is en telkens extra geheugen gealloceerd wordt wanneer dit nodig is. Dient de interface in het headerbestand hiervoor aangepast te worden?

Deel III

Software Ontwikkeling in C++

Hoofdstuk 10

C++ als uitbreiding van C

10.1 C++ historiek

In 1979 startte Bjarne Stroustrup een project in de Bell Laboratories, VS met als naam *C with classes*. De doelstelling van dit project was om klassen toe te voegen als een extensie aan C zonder run-time efficiëntie op te offeren. Het werd geïmplementeerd als een preprocessor voor C dat C met klassen converteerde in C code, welke vervolgens gecompileerd werd door een ANSI C compiler. In 1980 resulteerde dit in het eerste onderzoeksartikel met als titel: *C with classes*.

In 1982 werd het project uitgebreid door toevoeging van virtuele functies (en ondersteuning voor polymorfisme), overloading, referenties, en verbeterde type controle. Door deze vele uitbreidingen werd de taal nu C++ genoemd. In 1985 werd door de Bell Laboratories de *Cfront* compiler uitgebracht. Deze compiler vertaalt C++ in C om overdraagbaarheid (Eng.: portability) te ondersteunen op een groot aantal verschillende platformen.

Standardisatie-inspanningen begonnen in 1990 (ANSI/ISO C++ standardisatie comité). Sjablonen (Eng.: templates) en exceptions werden in 1990 ook aan de taal toegevoegd. Naamruimten (Eng.: name spaces) en RTTI (Eng.: Runtime Type Information) werden toegevoegd in 1993. ISO 14882, gepubliceerd in September 1998, is het eerste standar-

dizatie document voor C++.

De firma Borland bracht de eerste commerciële C++ compiler uit in 1990 en Microsoft in 1992.

10.2 Van abstract datatype naar klasse

Een belangrijke vaststelling in C is dat zelf-gedefinieerde types niet dezelfde faciliteiten krijgen als ingebouwde types. Het zou handig zijn dat:

1. er automatische creatie is van zelf-gedefinieerde types: zodat er niet expliciet geheugen hoeft voor gereserveerd te worden door oproep van bijvoorbeeld een `init` of `create` functie.

2. er automatische vernietiging is van zelf-gedefinieerde types wanneer deze out-of-scope gaan: zodat het ingenomen geheugen niet explicet dient vrijgegeven te worden door oproep van bijvoorbeeld een `destroy` functie.
3. het copieergedrag van zelf-gedefinieerde types wijzigbaar is: zodat er bij gebruik van de `=operator` (assignatie-operator of toekenning-operator) niet steeds een ondiepe kopie (Eng.: shallow copy) genomen wordt, maar men kan zorgen dat er een diepe kopie (Eng.: deep copy) genomen wordt.

Deze drie eigenschappen zijn niet mogelijk in C. De abstracte datatypes (ADT's), die in het deel C van deze cursus aan bod kwamen, hebben volgende tekortkomingen:

1. alle functies kunnen steeds opgeroepen worden door de gebruikers (i.e. geen mogelijkheid om de functies `public`, `private` of `protected` te maken, zoals in Java). Zoals vermeld bij statische functies (Eng.: static functions), kunnen deze wel gebruikt worden om de toegang tot een functie te beperken tot een compilatie-eenheid (i.e. een bestand). Deze kunnen niet in combinatie met een headerbestand gebruikt worden.
2. het is de verantwoordelijkheid van de gebruiker om de functies voor het geheugenbeheer (Eng.: memory management), i.e. creatie en vernietiging, op het gepaste moment op te roepen.
3. de functies zijn niet explicet verbonden met de data: de data is gesstructureerd in een `struct` en de functies staan hier los van (bij een functieoproep dient steeds de `struct` of het adres van de `struct` als argument meegegeven te worden).
4. datatypes kunnen niet geparametriseerd worden: bij een datatype is een keuze gemaakt om de gegevens op te slaan als bijvoorbeeld `int`'s, `double`'s, `char*`'s of zelf gedefinieerde `struct`'s, etc. Deze keuze is echter vast en indien men de gegevens als een ander type wil opslaan, dient de implementatie opnieuw te gebeuren.

In C++ is tegemoet gekomen aan deze beperkingen:

1. functies in een klasse worden methoden van deze klasse genoemd en deze kunnen `public`, `private` of `protected` aangeduid worden, net zoals in Java.
2. aan de hand van de constructor en destructor kunnen de functies voor het geheugenbeheer (creatie en vernietiging) automatisch opgeroepen worden.
3. de methoden zijn een explicet onderdeel van een klasse en dus verbonden met de data van een klasse. De data-variabelen van een klasse worden attributen genoemd.
4. datatypes kunnen geparametriseerd worden: door gebruik van sjablonen (Eng.: templates).

C++ is net zoals C gebaseerd op het principe van headerbestanden, waarin de declaraties van klassen (met bijhorende methoden en attributen), functies en globale variabelen voorkomen. In tegenstelling tot Java, is men in C++ niet verplicht om alle variabelen en functies in een klasse te plaatsen: men kan naast klassen ook functies en globale variabelen gebruiken (C-stijl) in combinatie met gebruik van deze klassen. Headerbestanden krijgen net zoals in C de extensie `*.h`. De `*.c` bestanden uit C worden in C++ vervangen door `*.cpp` bestanden. Deze bevatten de implementaties van de declaraties uit de headerbestanden enerzijds of C++ code met het gebruik van de klassen (geïnstantieerde objecten van deze klassen) anderzijds. Bij een `#include` opdracht hoeft de extensie `.h` van een headerbestand uit de C++ standaard bibliotheek niet expliciet vermeld te worden.

10.3 Online informatie

Twee interessante referenties, die de volledige taal C++ behandelen:

- cplusplus.com website: <http://wwwcplusplus.com/doc/tutorial/index.html>
- boek *Thinking in C++, Introduction to Standard C++*, Bruce Eckel, downloadbaar op <http://www.mindview.net/Books>

10.4 Naamruimten

Zoals vermeld dienen in C alle functienamen en globale variabelen in een project uniek te zijn (dus geen twee functies of globale variabelen met een identieke naam in alle bestanden van een project). In C++ kan men een naamruimte (Eng.:name space) aanmaken door volgende syntax:

```
namespace <naam>
{
}
```

met `<naam>` de gekozen naam van de naamruimte. Binnen een naamruimte dienen alle functienamen en globale variabelen een unieke naam te hebben, maar deze kunnen wel met dezelfde naam voorkomen in een andere naamruimte. In volgend voorbeeld worden twee naamruimten met naam A en B aangemaakt en komt de functie `f` in de beide naamruimten voor:

```
namespace A{
...
int f(int){...}
...
}
namespace B{
...
```

```
int f(int){...}
...
}
```

Deze code kan in hetzelfde bestand voorkomen, maar meestal wordt een nieuwe naamruimte in een apart bestand geplaatst. Wanneer men dan de functie `f` wil oproepen, dient men aan de hand van de *scope resolution operator* (`::`) aan te geven welke naamruimte men wil gebruiken, bijvoorbeeld:

```
void main(){
    A::f(7);
    B::f(8);
}
```

In de standaardbibliotheek van C++ is de naam van de naamruimte `std`. Om te vermijden dat bij veelvuldig gebruik van functies, objecten of variabelen uit een naamruimte men steeds `<naam>::` voor het gebruik dient te vermelden, kan men ook `using namespace <naam>;` in een bestand gebruiken: de naamruimte `<naam>` is dan de default naamruimte in dit bestand en `<naam>::` hoeft niet telkens explicet vermeld te worden. De opdracht `using namespace std;` treft men dan ook vaak bovenaan in bestanden aan waar veelvuldig van de standaard bibliotheek gebruik gemaakt wordt.

Bemerkt dat een `using namespace <naam>;` opdracht niet thuishoort in een headerbestand: men verplicht de gebruiker van het headerbestand om de naamruimte `<naam>` te gebruiken. `using namespace <naam>;` opdrachten horen thuis in de cpp-bestanden.

10.5 Output en input in C++: I/O streams

In C wordt er gebruik gemaakt van `printf` en `scanf`. In C++ worden deze echter **niet** meer gebruikt en dit omwille van twee redenen:

1. de compiler kan niet controleren of de argumenten wel degelijk zinvol in de formaat-string kunnen gesubstitueerd worden, bijvoorbeeld `printf("x=%s\n", x);` waarbij `x` een struct variabele is, kan niet gecontroleerd worden of dit wel zinvol is.
2. `printf` en `scanf` zijn niet uitbreidbaar voor gebruikergedefinieerde types: het aantal conversie-karakters in de formaat-string van `printf` en `scanf` is voorgedefinieerd en kan niet aangepast worden. Een `struct` kan niet ineens afgedrukt worden, maar elk veld apart op voorwaarde dat dit overeenkomt met een voorgedefinieerde formaat specifier.

Daarom worden beide functies in C++ vervangen door respectievelijk de `<<` operator en de `>>` operator.

10.5.1 Output: `<<` operator

Beschouw het volgende programma:

```
// Hello World programma in C++
#include <iostream>
using namespace std;

int main() {
    int a=7;
    cout << "Hello World" << endl;
    int b=12;
    cout << a << "+" << b << " = " << a+b << endl;
    return 0;
}
```

Commentaar van één regel kan in C++ aan de hand van // aangeduid worden. Input en output gebeurt in C++ steeds door middel van stream objecten: `cout` (afkorting van console output) is een object van de `ostream` (afkorting van output stream) klasse. Om dit object te kunnen gebruiken dient men `iostream` te includeren. Vermits de naamruimte van dit headerbestand `std` is, is de opdracht `using namespace std;` zeer handig (om te vermijden dat men bijvoorbeeld telkens `std::cout` dient te schrijven ipv `cout`). Men kan een stroom van gegevens naar dit `cout` object sturen (vandaar de naam *stream*), en het afdrukken gebeurt in dezelfde volgorde als waarin de gegevens doorgestuurd werden.

De operator `<<` wordt gebruikt om de gegevens door te geven (ipv een functie). De inspiratie voor deze operator komt van Unix shell commando's, waar deze operator ook gebruikt wordt. `endl` heeft dezelfde betekenis als '\n' (Eng.: new line).

Variabelen van fundamentele datatypes kunnen rechtstreeks in combinatie met de `<<` operator gebruikt worden. Voor zelf gedefinieerde klassen, dient de `<<` operator in de klasse declaratie en definitie voorzien te worden (via operator overloading zoals verder in hoofdstuk 12 aan bod komt).

De `<<` operator is concateneerbaar (i.e. kan verschillende malen na elkaar in dezelfde uitdrukking gebruikt worden, zoals in het voorbeeld hierboven). Controle-informatie kan via een *manipulator* aan de output stream doorgegeven worden: 3 voorbeelden van manipulatoren zijn: `oct` (i.e. druk bij elk volgend gebruik van `<<` de gegevens in octaal stelsel af), `dec` (i.e. druk bij elk volgend gebruik van `<<` de gegevens in decimaal stelsel af) en `hex` (i.e. druk bij elk volgend gebruik van `<<` de gegevens in hexadecimaal stelsel af).

10.5.2 Input: `>>` operator

Voor input wordt het `cin` (afkorting van console input) object gebruikt van de `istream` (afkorting van input stream) klasse, eveneens uit de `iostream` bibliotheek. Gebruik van dit object vereist dus eveneens `#include <iostream>`. De returnwaarde van de `>>` operator is `true` als het inlezen gelukt is en `false` als het inlezen mislukt is (vergelijkbaar met `scanf`). Volgend voorbeeld illustreert het gebruik van `cin`, `>>`, `<<` en de `hex` manipulator:

```
#include <iostream>
```

```

using namespace std;

int main() {
    int i=0;
    cout << "Geef getallen in :\n";
    while(cin>>i) {
        cout << "getal dec = " << i;
        cout << "\tgetal hex = " << hex << i << endl;
    }
    cout << "Einde ...";
    return 0;
}

```

10.5.3 Manipulatie van bestanden in C++

Bestanden worden in C++ eveneens via streams behandeld: **ofstream** klasse (afkorting van output file stream) voor output (i.e. schrijven van gegevens) naar een bestand en **ifstream** klasse (afkorting van input file stream) voor input (i.e. lezen van gegevens). Gebruik van beide klassen vereist **#include <fstream>**.

In volgend programma illustreert de functie **schrijfRandom** het openen, schrijven en sluiten van een bestand in C++ en de functie **leesRandom** het openen, lezen en sluiten van een bestand in C++:

```

#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;

void schrijfRandom(int);
void leesRandom();

int main() {
    schrijfRandom(10);
    leesRandom();
    return 0;
}

void schrijfRandom(int n) {
    ofstream out("random.txt");
    for(int i=0;i<n;i++)
        out << rand() << endl;
    out.close();
}

void leesRandom() {
    int r;
    ifstream in("random.txt");
    while(!in.eof()) {
        in >> r;
    }
}

```

```

        cout<< "getal = "<<r<<endl;
    }
    in.close();
}

```

Op een `fstream` object kan ook volgende methode toegepast worden voor het openen van een bestand:

```
void open (const char * filename, openmode mode);
```

Het tweede argument `mode` van deze methode kan hierbij de waarden uit tabel 10.1 aannemen.

<code>ios::in</code>	open bestand om te lezen
<code>ios::out</code>	open bestand om te schrijven
<code>ios::ate</code>	initiële positie: einde van het bestand (ate: at the end)
<code>ios::app</code>	alle gegevens worden telkens achteraan het bestand toegevoegd
<code>ios::trunc</code>	als het bestand reeds bestaat, wordt het eerst gewist
<code>ios::binary</code>	binaire mode

Tabel 10.1: Mogelijke waarden van `mode` bij openen van een `fstream`

10.6 C++ types

10.6.1 Fundamentele types

C++ kent dezelfde fundamentele datatypes als C, inclusief het gebruik van `short/long` en `signed/unsigned`. Een nieuw type is `bool` (vergelijkbaar met `boolean` in Java). Variabelen van het type `bool` kunnen als waarden ofwel `true` ofwel `false` aannemen. Controlestructuren in C++ worden gestuurd door een uitdrukking van het type `bool` (net zoals in Java waarin controlestructuren gestuurd worden door uitdrukkingen van het type `boolean`). Om compatibiliteit met C te garanderen, gebeurt er waar nodig automatische conversie van een variable van het type `bool` naar een `int` (hierbij wordt `true` dan omgezet naar 1 en `false` naar 0).

10.6.2 enum in C++

`enum` kan net zoals in C gebruikt worden om de mogelijke waarden van een variabele op te sommen. Echter in C++ definieert een `enum` een nieuw type (en is niet equivalent met het `int` type zoals in C). Variabelen van een verschillend `enum` type kunnen dus niet zomaar aan elkaar gelijkgesteld worden of incrementering (`++`) of decrementering (`--`) kan in C++ niet op een variabele van een `enum` type toegepast worden, zoals geïllustreerd in volgende code:

```

typedef enum dag_ {ma,di,wo,do,vr,za,zo} dag;
typedef enum seizoen_ {lente,zomer,herfst,winter} seizoen;
...
seizoen s=winter;
dag d=ma;
...
d=s; // OK in C, niet in C++ !!
d++; // idem

```

10.6.3 Type casting

Om het resultaat van een uitdrukking om te zetten naar een bepaald type, kan men dit in C++ doen op drie verschillende manieren, die alle drie tot hetzelfde resultaat leiden:

- `(type)uitdrukking`: operator-syntax
- `type(uitdrukking)`: functieoproep-syntax
- `static_cast<type>(uitdrukking)`: C++ expliciete cast, met template-syntax zoals in hoofdstuk 14 behandeld wordt

Een voorbeeld waarbij een variabele van het type `long` omgezet wordt naar een `int` op de drie verschillende maar equivalente manieren:

```

long l;
int i;
i=(int)l;
i=int(l);
i=static_cast<int>(l);

```

10.6.4 Referenties

In C++ komt met elk type (zowel een fundamenteel type, zelf gedefineerd `enum` type als een klasse) een geassocieerd referentiotype overeen. De syntax van dit type is `type&`. Een variabele van het referentiotype wordt een referentie genoemd. Deze bevat een adres van een variabele en wordt automatisch gederefereneerd wanneer deze in een uitdrukking gebruikt wordt (de dereferentie operator `*` hoeft dus niet opgeroepen te worden). Er is dus een sterke gelijkenis met wijzertypes (Eng.: pointer types). De twee belangrijke verschillen met pointers zijn echter:

1. op referenties kan geen aritmetiek (e.g. `++`, `+=4`, etc.) toegepast worden
2. een NULL referentie bestaat niet, een variabele van het referentiotype dient verplicht geïnitialiseerd te worden.

Referentiotypes laten een eenvoudiger syntax toe dan wijzertypes, zoals geïllustreerd in volgend voorbeeld. Hierbij wordt de functie `bepaalSom` gedeclareerd, geïmplementeerd en oproepen voor 2 gevallen: (i) met een wijzertype (`bepaalSomWijzer`) en (ii) met een

referentietype (`bepaalSomRef`). Beide functies leveren exact hetzelfde resultaat op. De syntax van de functie `bepaalSomRef` is echter eenvoudiger (zowel voor implementatie en oproep). Daarom wordt in C++ zeer vaak geopteerd voor functie-signaturen met referentie-syntax.

```
#include <iostream>
using namespace std;

void bepaalSomWijzer(int,int,int*);
void bepaalSomRef(int,int,int&);

int main(void) {
    int x,y,z;
    x=3;y=4;z=0;
    bepaalSomWijzer(x,y, &z);
    cout<<"De som is : "<<z<<endl;
    x=3;y=5;z=0;
    bepaalSomRef(x,y,z);
    cout<<"De som is : "<<z<<endl;
    return 0;
}

void bepaalSomWijzer(int a,int b,int* c) {
    *c=a+b;
}

void bepaalSomRef(int a,int b,int& c) {
    c=a+b;
}
```

10.7 Default functie-argumenten

In C++ kan je functie-argumenten bij declaratie een waarde geven. Een dergelijk argument vereist geen extra implementatie en zal, wanneer het argument niet meegegeven wordt bij een functie-oproep, de opgegeven default waarde krijgen.

Onderstaande code illustreert het gebruik van een default argument:

```
int som(int x, int i=101) {
    return x + i;
}
```

Bij volgende functie-oproepen levert dit het resultaat in commentaar op:

```
int j;
j=som(99,1); // j=100
j=som(99); // j=200
```

10.8 Dynamisch geheugenbeheer in C++

Voor dynamische allocatie van geheugen worden in C de functies `malloc`, `calloc` en `realloc` gebruikt. In C++ worden deze vervangen door de `new` operator. Het is nog steeds, net zoals in C, de verantwoordelijkheid van de programmeur om voldoende geheugen te reserveren vooraleer een dynamische variabele gebruikt wordt. De geheugenstructuur (text of code segment, stack segment, heap segment) zoals uitgelegd bij dynamisch geheugenbeheer in C is identiek in C++. De 7 regels en illustratieve scenario's uit appendix A zijn nog steeds relevant en zeer belangrijk! Enkel de syntax verschilt in C++ ten opzichte van C. De syntax komt hieronder aan bod.

Wanneer de `new` operator gebruikt wordt zonder vierkante haakjes (i.e. `[]`), wordt geheugen gealloceerd voor een enkelvoudige variable op te slaan. Wanneer de `new` operator gebruikt wordt met vierkante haakjes duidt dit op de allocatie van een array van variabelen.

Net zoals in C is het de verantwoordelijkheid van de programmeurs in C++ om expliciet dynamisch gealloceerd geheugen weer vrij te geven wanneer dit niet meer vereist is. In C dient hiervoor de `free` functie opgeroepen te worden, terwijl in C++ de `delete` operator hiervoor gebruikt wordt. Het is belangrijk om op te merken dat gebruik van een `delete` zonder vierkante haakjes het geheugen van een enkelvoudige variabele vrijgeeft, terwijl `delete` met vierkante haakjes een array van variabelen vrijgeeft.

Beschouw volgend voorbeeld in C met dynamische allocatie van een array `d`:

```
int main(){
    double* d;
    int i;
    int nr_doubles;
    /* read the number of values from file, via keyboard input or
       as a command line argument via argv and store this value in
       nr_doubles */
    d=(double*)malloc(nr_doubles*sizeof(double));

    /* read the values either (1) from file, (2) via keyboard input
       or (3) as command line arguments via argv and store them in the
       d array*/

    for(i=0;i< nr_doubles;i++){
        printf("d[%d]=%f\n",i,d[i]); /* print d[i] */
    }
    free(d);
    return 0;
}
```

De C++ versie van het bovenstaande voorbeeld wordt hieronder weergegeven:

```
int main(){
```

```
double* d;
int i;
int nr_doubles;
/* read the number of values from file, via keyboard input or
as a command line argument via argv and store this value in
nr_doubles */

d = new double[nr_doubles];

/* read the values from file, via keyboard input or as command
line arguments via argv*/

for(i=0;i< nr_doubles;i++){
    cout << "d[" << i << "]=" << d[i] << endl; //print d[i]
}
delete[] d; // en NIET delete d; !
return 0;
}
```

Belangrijk: gebruik van `malloc`, `calloc`, `realloc` en `free` in C++ is sterk afgeraden: gebruik **altijd** `new` en `delete` in C++. Een equivalent voor `realloc` bestaat **niet** in C++: de enige oplossing is de creatie met `new` van een nieuwe array, bestaande waarden eventueel overkopiëren en de oude array vrijgeven met `delete []`.

Hoofdstuk 11

Klassen in C++

11.1 C++ programma ontwerp

Er kunnen vier stappen onderscheiden worden bij het ontwerp van een programma in C++. Deze komen hieronder één voor één aan bod.

11.1.1 Interface specificatie

In deze stap worden de nodige klassen gedeclareerd in één of meerdere headerbestanden. Beschouw volgend voorbeeld, het headerbestand `Greeting.h` waarin de klasse `Greeting` gedeclareerd wordt:

```
// File:           $Id$  
// Description: Print Greeting Message  
// Revisions:  
//   $Log$  
  
#ifndef GREETING_H  
#define GREETING_H  
  
class Greeting {  
public:  
    void hello();  
}; // class Greeting  
  
#endif
```

Bemerkt dat de naam van het bestand niet hoeft overeen te komen met de naam van de klasse (in tegenstelling tot Java, waar dit wel het geval is), maar voor de duidelijkheid is dit meestal handig. De eerste vier regels van dit bestand bevatten commentaar, die niet door de compiler verwerkt worden. In deze commentaar zijn CVS (Eng.: Concurrent Version System) sleutelwoorden aanwezig: `Id` en `Log`. Deze worden door de CVS server automatisch ingevuld: `Id` wordt geëxpandeerd met de bestandsnaam, het versienummer, de datum van laatste wijziging en de naam van de auteur, `Log`

wordt geëxpandeerd met de log boodschap van de programmeur die de laatste versie heeft geüpload (en bevat een korte beschrijving van de aangebrachte wijzigingen in het bestand). Het gebruik van CVS is handig wanneer een project gemaakt wordt, waarbij verschillende programmeurs betrokken zijn die aan dezelfde code werken. Het gebruik van CVS is niet enkel op C++ gericht, maar is ook nuttig bij andere programmeertalen (zoals C of Java). We verwijzen naar hoofdstuk 17 voor een overzicht van CVS.

De volgende regels bevatten dan de `#ifndef #define` constructie: gebruik van de pre-processor voor conditionele compilatie. Deze conditionele compilatie in C++ is identiek aan C en zorgt ervoor dat bij expansie van de `#include` opdrachten in een bestand, elk headerbestand slechts éénmaal geïncludeerd wordt.

Vervolgens komt de eigenlijke declaratie van de klasse aan bod: sleutelwoord `class` gevolgd door de gekozen naam van de klasse en tussen accolades de methoden (functies) en de attributen (data variabelen) van de klasse. De accolades dienen afgesloten te worden met ; (in tegenstelling tot Java). Het sleutelwoord `public` geeft aan welke methoden op objecten van deze klasse extern kunnen opgeroepen worden. In tegenstelling tot Java dient het `public` sleutelwoord niet voor elke methode-declaratie herhaald te worden, maar gebruik van `public`: geeft aan dat alle volgende methoden de eigenschap `public` hebben. Andere sleutelwoorden om de toegang tot een methode te bepalen zijn: `private` (enkel de methoden van de klasse zelf kunnen deze methode oproepen) of `protected` (komt in het hoofdstuk 13 aan bod).

De methoden van deze klassen worden gedeclareerd in dit headerbestand: de volledige signatuur (methode naam, type return waarde, aantal argumenten, volgorde en type van de argumenten) wordt vastgelegd. De implementatie van deze methoden gebeurt in de volgende stap.

11.1.2 Klasse implementatie

De implementatie van de klasse (i.e. de implementatie van de gedeclareerde methoden) gebeurt in een `*.cpp` bestand. Hierin wordt het headerbestand met de interface specificatie (klasse declaratie) geïncludeerd. In volgend voorbeeld wordt het bestand `Greeting.cpp` getoond:

```
#include <iostream>
using namespace std;

#include "Greeting.h"

void Greeting::hello() {
    cout << "Hello, world!" << endl;
}
```

Hierin wordt de methode `hello` van de klasse `Greeting` geïmplementeerd. `#include <iostream>` zorgt dat gebruik van `cout` mogelijk wordt. De exacte signatuur van de methode wordt herhaald en vóór de naam van de methode wordt `klasse_naam::` vermeld: elke klasse definieert een naamruimte met als naam de naam van de klasse (op

dezelfde methode-naam in verschillende klassen voorkomen). Vervolgens wordt de eigenlijke code van de methode ingevuld.

11.1.3 Applicatie programma

Eens een klasse gedeclareerd en geïmplementeerd is, kan deze gebruikt worden: er kunnen objecten van geïnstantieerd worden en de publieke methoden op deze objecten kunnen opgeroepen worden. Dit wordt geïllustreerd in volgend bestand `SayHello.cpp`:

```
#include "Greeting.h"

int main() {
    Greeting greeting;
    greeting.hello();
    return 0;
}
```

Een object `greeting` van de klasse `Greeting` wordt aangemaakt en op dit object wordt de methode `hello()` opgeroepen. Bemerk dat de namen van de objecten volledig onafhankelijk van de klassenamen kunnen gekozen worden (net zoals in Java).

11.1.4 Compilatie en linking

Volgende stap is uiteraard de compilatie van de bestanden en het linken van de gecompileerde bestanden tot één uitvoerbaar bestand. Deze stap is volledig identiek met C: bestanden worden elk apart gecompileerd en de linker zorgt ervoor dat verwijzingen tussen bestanden opgelost worden en dat de nodige code uit de bibliotheken ingevoegd wordt. In ons voorbeeld resulteert compilatie in twee object-bestanden: `Greeting.o` en `SayHello.o`, welke door de linker dan omgezet worden in een uitvoerbaar bestand met als naam bijvoorbeeld `SayHello`.

11.2 Constructoren en destructor

Een constructor en een destructor zijn speciale methoden van een C++ klasse: ze worden automatisch opgeroepen bij creatie van een object van deze klasse (constructor) en bij het *out of scope* gaan van een object (destructor). Ze zorgen er dus voor dat er door de gebruiker van de klasse geen expliciete `create` of `init` methoden of `destroy` methoden dienen opgeroepen te worden. Een constructor zorgt voor een correct geïnitialiseerd object en een destructor vermijdt geheugenlekken. Het is echter de verantwoordelijkheid van de programmeur om correcte constructoren en een destructor te voorzien.

Een constructor heeft steeds **exact** dezelfde naam als de klasse zelf en de destructor heeft ook exact dezelfde naam, maar met een ~(tilde) voor. Een klasse kan meerdere constructoren hebben (met telkens andere argumenten), maar slechts één destructor. Constructoren en destructoren hebben **geen** return waarde (vermits ze niet expliciet opgeroepen worden). Een klasse `A` heeft als destructor `~A()` en bijvoorbeeld als constructoren `A()` (zonder argumenten, wordt default constructor genoemd), `A(int)` (met

één geheel argument), `A(std::string, int i)` (met twee argumenten: een string en een geheel getal).

Bemerkt dat constructoren en destructor steeds `public` dienen gedeclareerd te worden. Een belangrijke constructor is de kopie constructor (Eng.: copy constructor). Een kopie constructor van de klasse A heeft als syntax: `A(const A& a)`. Als argument wordt aan de constructor een referentie van een object van de klasse A meegegeven en dit argument wordt gebruikt om het object in kwestie te initialiseren (i.e. een kopie te nemen van het argument om het object te initialiseren). Er wordt een referentie gebruikt voor efficiëntie-redenen: in plaats van het object door te geven wordt enkel het adres doorgegeven, hetgeen compacter is en dus sneller kan uitgevoerd worden, `const` zorgt ervoor dat de constructor de waarde van het argument niet kan wijzigen.

In volgend voorbeeld wordt de declaratie van de klasse `Point` getoond, waarin twee constructoren voorkomen, een constructor met twee argumenten en een kopie constructor (bestand `point.h`):

```
#ifndef POINT_H
#define POINT_H

class Point {
public:
    Point(float x, float y);
    Point(const Point &p);
public:
    void set_x( float x_coord );
    void set_y( float y_coord );
    float get_x();
    float get_y();
private:
    float radius;
    float angle;
};

#endif
```

In het bestand `point.cpp` worden dan de implementaties voorzien van onder andere deze beide constructoren:

```
#include <math.h>
#include "Point.h"

Point::Point(float x_coord, float y_coord) {
    radius = sqrt(x_coord * x_coord + y_coord * y_coord);
    angle = atan2(y_coord, x_coord);
}

Point::Point(const Point& p) {
    radius = p.radius ;
    angle = p.angle;
}
```

Bemerk dat er in dit voorbeeld geen destructor nodig is (vermits er geen dynamisch gealloceerd geheugen is en er dus ook geen gebruikt geheugen hoeft vrijgegeven te worden). Volgend voorbeeld toont een klasse waarin er wel een destructor aanwezig is (deze geeft echter ook geen geheugen vrij, maar drukt een boodschap af via `cout`).

```
class MyClass {
private :
    int i;
public :
    MyClass(string s,int i);
    ~MyClass();
};

MyClass::MyClass(string s,int i) {
    cout << "Hallo, constructor van object "<<s<<endl;
    this->i=i;
}

MyClass::~MyClass(){
    cout << "Destructor van object met waarde : "<<i<<endl;
}
```

Bemerkt het gebruik van `this`: dit is een wijzervariabele, die telkens het adres van het huidige object bevat. `this->i` wordt hier gebruikt om het onderscheid te kunnen maken tussen het argument `i` en het attribuut `i` van de klasse: `this->i` is het attribuut en `i` het argument.

Wanneer een object van de klasse in volgende `main()` functie gebruikt wordt, wordt eerst de constructor en vervolgens de destructor opgeroepen:

```
#include <iostream>
#include <string>
#include <Myclass.h>
using namespace std;

int main() {
    MyClass a("a",1);
    return 0;
}
```

Volgende code illustreert het aanmaken van 8 objecten van deze klasse:

```
int main() {
    MyClass a("a",1);
    for(int i=0;i<3;i++) {
        MyClass b("b",i);
    }
    for(int j=0;j<3;j++) {
        MyClass c("c",10);
    }
}
```

```

int k=0;
for(MyClass d("d",20);k<3;k++) {
    cout << k << endl;
}
return 0;
}

```

Uitvoering van dit programma resulteert in volgende output (de volgorde en inhoud van de output geeft aan wanneer de objecten aangemaakt en verwijderd worden):

```

Hallo, constructor van object a
Hallo, constructor van object b
Destructor van object met waarde : 0
Hallo, constructor van object b
Destructor van object met waarde : 1
Hallo, constructor van object b
Destructor van object met waarde : 2
Hallo, constructor van object c
Destructor van object met waarde : 10
Hallo, constructor van object c
Destructor van object met waarde : 10
Hallo, constructor van object c
Destructor van object met waarde : 10
Hallo, constructor van object d
0
1
2
Destructor van object met waarde : 20
Destructor van object met waarde : 1

```

11.3 Initialisatie en allocatie van objecten

Er zijn twee manieren in C++ voor initialisatie en allocatie van objecten:

1. statische allocatie: een object wordt aangemaakt in het stack segment (cfr appendix A), bijvoorbeeld binnen een functie en wordt na oploop van de functie automatisch vrijgegeven.
2. dynamische allocatie: een object wordt aangemaakt in het heap segment (cfr appendix A), aan de hand van de **new** operator en dient nadien vrijgegeven te worden met **delete**.

Voorbeelden van statische allocatie zijn weergegeven in tabel 11.1 en voorbeelden van dynamische allocatie in tabel 11.2.

Volgend programma alloceert dynamisch een object:

```

#include <iostream>
#include <string>
using namespace std;

```

```
int main() {
    MyClass* a = new MyClass("a",1);
    return 0;
}
```

De destructor wordt hierbij NIET opgeroepen wanneer dit object out-of-scope gaat: geheugenlek (Eng.: memory leak)! In volgend programma wordt dit opgelost:

```
int main() {
    MyClass* a = new MyClass("a",1);
    delete a;
    return 0;
}
```

Het is dus zeer belangrijk dat de verantwoordelijkheid voor het opruimen van dynamisch gealloceerde objecten vastgelegd wordt!

<code>MyClass a;</code>	allocatie geheugen object met naam a, oproep constructor zonder argumenten voor a.
<code>MyClass a(7);</code>	allocatie geheugen object met naam a, oproep constructor met argumenten voor a.
<code>MyClass(7)</code>	allocatie geheugen voor (voorlopig) anoniem object, oproep van gepaste constructor.

Tabel 11.1: Voorbeelden van statische allocatie van objecten.

<code>MyClass* a = new MyClass();</code>	allocatie geheugen anoniem object, levert wijzer naar dit object, a bevat adres van het gecreëerde object, oproep constructor zonder argumenten.
<code>MyClass* a = new MyClass(7);</code>	allocatie geheugen anoniem object, idem als hierboven, met oproep gepaste constructor
<code>delete a;</code>	roept de destructor voor het object op.

Tabel 11.2: Voorbeelden van dynamische allocatie van objecten en vrijgave.

11.4 Friend functies en klassen

11.4.1 Friend functies

Een functie, vermeld in de declaratie van een klasse, kan in C++ als sleutelwoord **friend** krijgen. Dit betekent dat deze functie geen onderdeel is van de klasse (geen member-functie), maar wel toegang krijgt tot private attributen en methoden van deze klasse.

Met andere woorden: een **friend** functie heeft dezelfde rechten als member-functies. Beschouw volgende declaratie van de klasse **A** waarin de functie **g** als **friend** functie gedeclareerd wordt:

```
class A{
private:
    int i;
public:
    A();
    A(int i);
    int getAttribute();
    int f(int i);
    friend int g(A a);
};
```

De implementatie van de functie **g** is dan bijvoorbeeld als volgt:

```
int g(A a){ //geen A::g !
    cout << a.i; // rechtstreekse toegang tot privaat attribuut !
}
```

Dit voorbeeld toont dat de **friend** functie geen onderdeel is van de klasse (dus geen **A::g** in de implementatie) en dat er rechtstreekse toegang mogelijk is tot private attributen (**a.i** in dit voorbeeld) en private methoden van de klasse **A**.

Het voordeel van het gebruik van **friend** functies is dat men geen publieke methoden hoeft op te roepen om aan de waarden van private attributen te geraken (bijvoorbeeld oproepen van de methode **getAttribute()** in dit voorbeeld is niet nodig), wat dus zorgt voor schnellere uitvoering van het programma (zeker wanneer deze functies zeer veel worden opgeroepen).

11.4.2 Friend klassen

Volledige klassen kunnen in C++ ook als friend van elkaar worden aangeduid. Bijvoorbeeld een klasse **Y** is friend van klasse **Z**:

```
class Y{
    ...
}

class Z{
    ...
    friend class Y;
}
```

Dit betekent dat alle member-functies van de klasse **Y** friend zijn van de klasse **Z** (i.e. alle member-functies van de klasse **Y** hebben toegang tot de private attributen en private member-functies van de klasse **Z**). Deze eigenschap geldt echter **niet** omgekeerd (friend eigenschap wordt toegekend, niet genomen).

11.4.3 Friend concept in een object-georiënteerde taal

Volgens het stricte principe van object-oriëntatie (OO), dienen alle functies member-functies te zijn van een klasse. Het friend concept schendt eigenlijk de basisprincipes van object-oriëntatie!

Toch wordt het veel gebruikt in C++, vooral voor efficiëntie-verhoging bij operator overloading (zoals uitgelegd in volgend hoofdstuk).

Hoofdstuk 12: Overloading van Functies en Operatoren

Hoofdstuk 12

Overloading van Functies en Operatoren

12.1 Overloading van functies

Overloading (Ned.: overladig) van functies betekent dat de naam van een functie verschillende malen gebruikt wordt, maar met verschillend aantal argumenten of types van argumenten. Volgend voorbeeld toont een klasse met overloading van de methode `m`:

```
class A{
    public:
        int m(int i);
        int m(B b, int i);
        void m(B b, double d);
        bool m(B b);
};
```

In C++ en Java is overloading toegelaten, maar niet in C. Bemerk dat overloading op basis van de argumenten gebeurt: twee functies met identiek dezelfde argumenten (aantal, type en volgorde) maar met verschillend type van return waarde zijn **niet** toegelaten.

12.2 const methoden

Het sleutelwoord `const` kan toegevoegd worden aan een methode-declaratie: dit garandeert dat methode geen aanpassingen doet aan de attributen van een klasse. Gebeurt dit toch, dan geeft de compiler een fout en breekt de compilatie af.

Een belangrijk gebruik van constante methoden is in combinatie met constante objecten: op constante objecten kunnen enkel constante methoden opgeroepen worden. Volgend voorbeeld toont een klasse met constante methoden en het gebruik van een constant object:

```
class A{
    private:
        int i;
        double d;
```

```

public:
    A(int, double);
    int getValue_i() const;
    double getValue_d() const;
    void setValue_i(int);
    void setValue_d(double);
    void print() const;
};

...
const A a(1,4.0);
a.print(); // OK
int i = a.getValue_i(); //OK
a.setValue_i(2); // Compile error

```

12.3 Overloading van operatoren

12.3.1 Definitie en voorbeeld

Overloading van een operator (bijv. `+`, `-`, `=`, `==`, `<`, `++`, etc.) betekent dat men de operator declareert en implementeert voor gebruik op eigen gedefinieerde klassen. Volgend voorbeeld toont een `+` operator voor de klasse `A`:

```

class A{
    ...
    const A operator +(const A& a) const;
};

```

Deze kan op de volgende manier gebruikt worden:

```

A a1, a2, a3;
a3 = a1+a2;
// equivalent met:
// a3.operator=(a1.operator+(a2));

```

Het gebruik van de operator komt neer op een methode-oproep, waarbij de rechteroperand (object `a2` in dit voorbeeld) als argument doorgegeven wordt en de linkeroperand (object `a1` in dit voorbeeld) het object is, waarop de operator methode toegepast wordt. De linkeroperand wordt bij een `+` operator niet gewijzigd en kan dus als `const` referentie (`const A& a`) doorgegeven worden. Een `+` operator verandert ook niets aan de linkeroperand en dus kan de operator ook als `const` functie gedeclareerd worden (zoals beschreven in vorige sectie). Het resultaat van een `+` operator is de creatie van een nieuw object, dat de som bevat van de andere twee objecten. Dit nieuwe object wordt gebruikt om toe te kennen aan het derde object (object `a3` in dit voorbeeld): het heeft geen zin om dit object (`a1+a2`) aan te passen en het type van de return waarde kan dus als `const` gedeclareerd worden. Als type van de return waarde kan dus voor de `+` operator een `const A` object genomen worden.

Beschouw volgend voorbeeld waarbij men de `+`, `=` en `<` operator declareert en implementeert voor de klasse `Vector`:

```

#include <iostream>
using namespace std;

class Vector {
public:
    int x,y;
    Vector() {};
    Vector(int,int);
    const Vector operator+ (const Vector&) const;
    Vector& Vector::operator= (const Vector& param);
    bool Vector::operator< (const Vector& param) const;
};

Vector::Vector (int a, int b) {
    x = a;
    y = b;
}

const Vector Vector::operator+ (const Vector& param) const {
    Vector som;
    som.x = x + param.x;
    som.y = y + param.y;
    return (som);
}

Vector& Vector::operator= (const Vector& param) {
    x=param.x;
    y=param.y;
    return *this;
}

bool Vector::operator< (const Vector& param) const {
    return (x<param.x && y<param.y);
}

int main () {
    Vector a (3,1);
    Vector b (1,2);
    Vector c;
    c = a + b;
    cout << c.x << "," << c.y;
    return 0;
}

```

In dit voorbeeld zijn volgende punten belangrijk:

- de constructor `Vector()` is reeds geïmplementeerd in het headerbestand: implementaties van korte methoden gebeuren vaak in het headerbestand.
- als type van de return waarde van de `+` operator is `Vector` genomen en niet

Vector&: resultaat van een `+` is een nieuw object dat de som van de linker- en rechteroperand bevat (een referentie naar een bestaand object is dus niet mogelijk).

- bij de `=` operator is als type van de return waarde een `Vector&` vereist (en niet `Vector`): de `=` operator zorgt voor de aanpassing van de linkeroperand en men wil de `=` operator in cascade kunnen gebruiken. In cascade betekent dat de operator meerdere keren in dezelfde uitdrukking voorkomt (bijv. `a = b = c;`). Hiervoor is het belangrijk dat het aangepaste object als return waarde wordt meegegeven. De opdracht `return *this;` zorgt ervoor dat een referentie naar het object zelf als return waarde wordt meegegeven (`this` bevat steeds het adres van het object in kwestie).
- indien de `=` operator niet expliciet gedeclareerd en geïmplementeerd wordt, wordt deze door de compiler automatisch gegenereerd (en wordt telkens een ondiepe kopie van de attributen genomen). Vooral om ervoor te zorgen dat steeds een diepe kopie genomen wordt, is het overladen van de `=` operator zeer belangrijk.
- de operatoren `+` en `<` kunnen `const` functies zijn omdat ze niets veranderen aan de attributen van het object waarop ze toegepast worden. De `=` operator kan uiteraard niet als `const` functie gedeclareerd en geïmplementeerd worden.

12.3.2 Drie opties voor declaratie en implementatie

Volgende gevallen in kader van operator overloading kunnen onderscheiden worden:

1. de operator is als klassemember gedeclareerd en heeft dus bijgevolg toegang tot de private attributen van deze klasse. De linkeroperand is telkens het object zelf, en de rechteroperand is het argument van de operator.
2. de operator is buiten de klasse gedeclareerd en heeft dus enkel toegang tot de private attributen van een klasse via publieke methoden (bijv. `set/get` methodes). De linkeroperand is dan het 1e argument van de operator, en de rechteroperand is het 2e argument van de operator.
3. de operator is als `friend` in een klasse gedeclareerd, en is dus geen klassemember, maar heeft wel rechtstreekse toegang tot de private attributen van deze klasse. De linkeroperand is opnieuw het 1e argument van de operator, en de rechteroperand is het 2e argument van de operator.

Deze drie opties worden in sectie 12.3.6 geïllustreerd.

12.3.3 Overzichtstabel

Tabel 12.1 toont verschillende operatoren en hun signaturen, in geval de operator als klassemember is gedeclareerd (optie 1). Het is belangrijk om de keuze van deze signaturen zeer goed te begrijpen.

naam	return type	argument	const
+	const A	const A& a	ja
-	const A	const A& a	ja
==	bool	const A& a	ja
-(unair)	const A	(geen)	ja
=	A&	const A& a	nee
<	bool	const A& a	ja
++(prefix)	A&	(geen)	nee
+=	A&	const A& a	nee

Tabel 12.1: Voorbeelden van operatoren en hun signatuur

12.3.4 Postfix en prefix operatoren

De prefix en postfix operator worden beiden door het `++` symbool voorgesteld. Er dient dus een onderscheid gemaakt te worden zodat de compiler de juiste keuze kan maken. Men heeft voor volgende conventie gekozen:

- de prefix operator als klassemember heeft geen argumenten,
- de postfix operator als klassemember heeft een `int` als argument: de waarde van dit geheel getal wordt niet gebruikt in de implementatie, maar dient zuiver om het onderscheid te maken met de prefix operator. Het is ook niet nodig om dit getal door te geven bij het gebruik van de postfix operator.

Volgend voorbeeld toont een klasse A met prefix en postfix operatoren (zowel increment als decrement).

```
class A {
public:
    A& operator++(); //Prefix increment operator (++x)
    A operator++(int); //Postfix increment operator (x++)
    A& operator--(); //Prefix decrement operator (--x)
    A operator--(int); //Postfix decrement operator (x--)
};
```

12.3.5 Andere operatoren

Volgende andere operatoren kunnen onderscheiden worden:

1. operatoren `*`, `%`, `<=`, `>`, `>=`, `-=`, `[]`: kunnen op een vergelijkbare manier als de operatoren hierboven gedeclareerd en geïmplementeerd worden.
2. operatoren `&&`, `||`: bij overladen van deze operator geldt de kortsluitingsregel (Eng.: short circuit rule) niet meer, het is dus sterk afgeraden om deze te overladen. De kortsluitingsregel bestaat erin dat de evaluatie van een uitdrukking stopt wanneer het resultaat bekend is.

3. de komma operator (,): bij overloaden van deze operator is er geen links naar rechts evaluatie meer (zoals vereist bij doorgeven van functie-argumenten): het is ook sterk afgeraden om deze te overloaden.
4. de **iostream** operatoren <<, >>: deze komen aan bod in volgende sectie (operator overloading met friends).

12.3.6 Code voorbeelden

Optie 1: operator als klassemember

```
class A
{
public:
    A();
    A(double d);
    A(int a1, int a2);
    A(int a1);
    int get_a1();
    int get_a2();
    ...
    const A operator+(const A& secondOperand) const;
    const A operator-(const A& secondOperand) const;
    bool operator==(const A& secondOperand) const;
    const A operator-() const;
private:
    int a1, a2;
};

...
const A A::operator+(const A& secondOperand) const
{
    int sum_a1, sum_a2;
    //calculate sum of a1, secondOperand.a1 and a2, secondOperand.a2
    ...
    return A(sum_a1, sum_a2);
}

const A A::operator-(const A& secondOperand) const
{
    int diff_a1, diff_a2;
    //calculate difference between a1, secondOperand.a1 and a2, secondOperand.a2
    ...
    return A(diff_a1, diff_a2);
}

bool A::operator==(const A& secondOperand) const
{
```

```

        return ((a1 == secondOperand.a1)
                && (a2 == secondOperand.a2));
    }

const A A::operator-( ) const
{
    return A(-a1, -a2);
}

```

Optie 2: operator buiten klasse

```

class A
{
public:
    A();
    A(double d);
    A(int a1, int a2);
    A(int a1);
    int get_a1();
    int get_a2();
    ...

private:
    int a1, a2;
};

const A operator+(const A& firstOperand, const A& secondOperand);

const A operator-(const A& firstOperand, const A& secondOperand);

bool operator==(const A& firstOperand, const A& secondOperand);

const A operator-(const A& firstOperand);

...

const A operator+(const A& firstOperand, const A& secondOperand) const
{
    int sum_a1, sum_a2;
    //calculate sum of firstOperand.get_a1(), secondOperand.get_a1()
    //and firstOperand.get_a2(), secondOperand.get_a2()
    ...
    return A(sum_a1, sum_a2);
}

const A operator-(const A& firstOperand, const A& secondOperand) const
{
    int diff_a1, diff_a2;
    //calculate difference between firstOperand.get_a1(), secondOperand.get_a1()
    //and firstOperand.get_a2(), secondOperand.get_a2()
    ...
}

```

```

        return A(diff_a1, diff_a2);
    }

bool operator==(const A& firstOperand, const A& secondOperand) const
{
    return ((firstOperand.get_a1() == secondOperand.get_a1())
            && (firstOperand.get_a2() == secondOperand.get_a2()));
}

const A operator-(const A& firstOperand) const
{
    return A(-firstOperand.get_a1(), -firstOperand.get_a2());
}

```

Optie 3: friend operator buiten klasse

```

class A
{
public:
    A();
    A(double d);
    A(int a1, int a2);
    A(int a1);
    int get_a1();
    int get_a2();
    ...

    friend const A operator+(const A& firstOperand, const A& secondOperand);
    friend const A operator-(const A& firstOperand, const A& secondOperand);
    friend bool operator==(const A& firstOperand, const A& secondOperand);
    friend const A operator-(const A& firstOperand);

private:
    int a1, a2;
};

...

const A operator+(const A& firstOperand, const A& secondOperand) const
{
    int sum_a1, sum_a2;
    //calculate sum of firstOperand.a1, secondOperand.a1
    //and firstOperand.a2, secondOperand.a2
    ...
    return A(sum_a1, sum_a2);
}

const A operator-(const A& firstOperand, const A& secondOperand) const
{
    int diff_a1, diff_a2;
    //calculate difference between firstOperand.a1, secondOperand.a1

```

```

//and firstOperand.a2, secondOperand.a2
...
return A(diff_a1, diff_a2);
}

bool operator==(const A& firstOperand, const A& secondOperand) const
{
    return ((firstOperand.a1 == secondOperand.a1)
            && (firstOperand.a2 == secondOperand.a2));
}

const A operator-(const A& firstOperand) const
{
    return A(-firstOperand.a1, -firstOperand.a2);
}

```

12.4 Operator overloading met friends

Een belangrijk voorbeeld van dit type operatoren zijn de `<<` en `>>` operatoren. Deze werken in op bijvoorbeeld het `cout` of `cin` object en kunnen dus per definitie niet als klassemember van een zelf-gedefinieerde klasse gedeclareerd worden. Ze dienen dus buiten de klasse gedeclareerd te worden en om rechtstreekse toegang te verkrijgen tot de private attributen kan deze operator als `friend` gedeclareerd worden. De return waarde is een `ostream&` (voor `<<`) of `istream&` (voor `>>`): om toe te laten dat een cascade van `<<` en `>>` operatoren in dezelfde uitdrukking mogelijk wordt (volledig vergelijkbaar met de `=` operator, zoals hierboven uitgelegd).

Volgend voorbeeld toont hetzelfde voorbeeld als hierboven maar nu met de `<<` en `>>` operatoren als `friend`:

```

class A
{
public:
    A();
    A(double d);
    A(int a1, int a2);
    A(int a1);
    int get_a1();
    int get_a2();
    ...
    const A operator+(const A& secondOperand) const;
    const A operator-(const A& secondOperand) const;
    bool operator==(const A& secondOperand) const;
    const A operator-() const;
    friend ostream& operator<<(ostream& outputStream, const A& secondOperand);
    friend istream& operator>>(istream& inputStream, A& secondOperand);
private:
    int a1, a2;
}

```

Hoofdstuk 12: Overloading van Functies en Operatoren

```
};

...

ostream& operator<<(ostream& outputStream, const A& secondOperand)
{
    //write secondOperand.a1 and secondOperand.a2 to outputStream
    outputStream << "a1=<< secondOperand.a1 << ":" << "a2=<< secondOperand.a2 << endl;

    return outputStream;
}

istream& operator>>(istream& inputStream, A& secondOperand)
{
    //read secondOperand.a1 and secondOperand.a2 from inputStream
    return inputStream;
}
```

Hoofdstuk 13

Overerving en Polymorfisme

13.1 Basisconcept overerving

Net zoals in Java, kan een klasse in C++ overerven van een basisklasse (Eng.: base class). Een nieuwe klasse erft over van een basisklasse, deze nieuwe klasse wordt de afgeleide klasse (Eng.: derived class) genoemd. De basisklasse kan aanziend worden als een algemene klasse, waarvan meer specifieke klassen zijn afgeleid. Een voorbeeld is een basisklasse **Voertuig** met afgeleide klassen **Auto** en **Bus**. De klasse **Auto** kan verder afgeleid worden tot **SportWagen**, **GezinsWagen** en **TerreinWagen**.

Een afgeleide klasse erft van de basisklasse de attributen en methoden (afhankelijk of ze **public**, **private** of **protected** gedeclareerd zijn, zoals verder uitgelegd wordt), en kan nieuwe attributen en methoden toevoegen of bestaande methoden opnieuw definiëren. Overerving (Eng.: inheritance) is een belangrijk principe van object-georiënteerd programmeren en komt in dit hoofdstuk uitgebreid aan bod.

13.2 Afgeleide klassen

Volgend voorbeeld toont de syntax van een basisklasse A en een afgeleide klasse B:

```
class A {  
public:  
    void setVarA(int i);  
    int getVarA();  
private:  
    int varA;  
};  
  
class B : public A {  
public:  
    void setVarB(int i);  
    int getVarB();  
private:  
    int varB;
```

```
};
```

De dubbele punt (:) na de klassenaam duidt op overerving, de naam van de basisklasse staat rechts en deze van de afgeleide klasse staat links. Het `public` sleutelwoord geeft aan dat publieke methoden in de basisklasse eveneens publiek worden in de afgeleide klasse. Private en protected overerving komen verder aan bod. Spaties voor en na de dubbele punt (:) mogen, maar hoeven niet.

Op objecten van de afgeleide klasse B kunnen zowel de publieke methoden van de basisklasse A als deze van de afgeleide klasse B opgeroepen worden, zoals hieronder geïllustreerd wordt:

```
B b;
b.setVarA(8);
b.setVarB(11);
```

13.2.1 Constructoren van afgeleide klassen

Constructoren van de basisklasse worden **niet** overgeërfd in de afgeleide klassen. Ze dienen opgeroepen te worden in de constructor van de afgeleide klasse. De regel is dat de constructor van de basisklasse alle attributen van de basisklasse dient te initialiseren en dat de constructor van de afgeleide klasse de constructor van de basisklasse oproept (in de initialisatie sectie, i.e. na de : bij de constructor implementatie). Vervolgens worden de attributen van de afgeleide klasse geïnitialiseerd.

Volgend code-fragment toont de declaratie van de basisklasse A en afgeleide klasse B, beiden met verschillende constructoren:

```
class A {
public:
    A();
    A(int i);
    void setVarA(int i);
    int getVarA();
private:
    int varA;
};

class B : public A {
public:
    B();
    B(int i);
    B(int varA, int varB);
    void setVarB(int i);
    int getVarB();
private:
    int varB;
};
```

De implementatie van deze constructoren wordt hieronder weergegeven:

```
A::A():varA(0) {}
A::A(int i):varA(i) {}
B::B():A(),varB(0) {}
B::B(int i):A(),varB(i) {}
B::B(int vara, int varb):A(vara),varB(varb){}
```

Indien de constructor van de afgeleide klasse geen constructor van de basisklasse oproept, wordt de default constructor van de basisklasse automatisch opgeroepen (de default constructor is de constructor zonder argumenten). Zeker in het geval van initialisatie van wijzerwaarden en allocatie van dynamisch geheugen is dit niet aangewezen!

13.2.2 Het sleutelwoord `protected`

Een afgeleide klasse heeft geen directe toegang tot private members (attributen en methoden) van de basisklasse, maar heeft enkel indirecte toegang tot private data via de publieke member-methoden. Private member-methoden zijn dus enkel bruikbaar in de klasse zelf, en niet in de afgeleide klassen. Private methoden zijn meestal hulpmethoden voor exclusief gebruik in de klasse zelf (en niet in de afgeleide klassen).

Het sleutelwoord `protected` in de declaratie van attributen en methoden van een klasse laat toegang toe tot de attributen en methoden in de afgeleide klassen. In de klasse waar een attribuut of methode als `protected` gedeclareerd is, heeft het attribuut of de methode dezelfde eigenschappen als was het `private` gedeclareerd. In een afgeleide klasse zijn de `protected` attributen en methoden van de basisklasse eveneens `protected`, zodat ze toegankelijk zijn voor verdere afleidingen.

13.2.3 Herdefinitie van member-methoden in afgeleide klassen

In de declaratie van een afgeleide klasse komen de declaraties van nieuwe methoden en attributen, maar er kunnen ook declaraties komen van overgeërfde methoden, die een nieuwe implementatie krijgen. Dit noemt men herdefinitie van methoden. Niet gedeclareerde overgeërfde member-methoden worden automatisch onveranderd overgeërfd. Herdefinitie is dus niet hetzelfde als overloading: bij overloading van methoden dienen de argumenten verschillend te zijn, bij herdefinitie blijft de signatuur van de methoden exact dezelfde.

Volgend voorbeeld toont de methode `print()` die in de afgeleide klasse `B` geherdefinieerd wordt:

```
class A {
public:
    A();
    A(int i);
    void setVarA(int i);
    int getVarA();
    void print();
private:
```

```

    int varA;
};

class B : public A {
public:
    B();
    B(int i);
    B(int varA, int varB);
    void setVarB(int i);
    int getVarB();
    void print();
private:
    int varB;
};

```

Op objecten van de klasse **A** wordt de **print()** methode van de klasse **A** opgeroepen en op objecten van de klasse **B** wordt de **print()** methode van de klasse **B** opgeroepen, tenzij aan de hand van **::** expliciet aangegeven wordt welke methode bedoeld wordt. Dit wordt hieronder geïllustreerd:

```

A a;
B b;

a.print();
b.print();
b.A::print();

```

In principe worden alle methoden overgeërfd in de afgeleide klasse, uitzonderingen op deze regel zijn:

1. constructoren, inclusief de copy constructor: indien deze niet gedeclareerd is, wordt er een default copy constructor aangemaakt (die telkens ondiepe kopieën maakt),
2. destructoren,
3. toekenningsoperator **=** operator: indien deze niet gedeclareerd is, wordt er een default **=** operator aangemaakt (die eveneens telkens ondiepe kopieën maakt).

Dus vooral wanneer wijzers gebruikt worden en allocatie van dynamisch geheugen gebeurt, is het belangrijk om bovenstaande methoden of operatoren correct te declareren en te implementeren in de afgeleide klassen. Deze komen in de volgende secties één voor één aan bod.

13.2.4 Copy constructor

Beschouw volgende declaratie van een basisklasse **A** en afgeleide klasse **B**, waarbij beiden een copy constructor hebben:

```

class A {
public:
    A();
    A(int i);
    A(const A& a);
    void setVarA(int i);
    int getVarA();
private:
    int varA;
};

class B : public A {
public:
    B();
    B(int i);
    B(int vara, int varb);
    B(const B& b);
    void setVarB(int i);
    int getVarB();
private:
    int varB;
};

```

De implementatie van de copy constructor in de afgeleide klasse B is belangrijk en wordt hieronder weergegeven:

```

B::B(const B& b): A(b){
    varB=b.varB;
}

```

De code toont dus eerst het oproepen van de constructor van de basisklasse (in de initialisatie-sectie) en vervolgens tussen accolades ({})) de initialisatie van de attributen van de afgeleide klasse.

13.2.5 Destructoren in afgeleide klassen

Wanneer de destructor van de afgeleide klasse opgeroepen wordt, roept deze **automatisch** de destructor van de basisklasse op. Deze laatste hoeft dus niet expliciet opgeroepen te worden. De regel is dus dat de destructor in een afgeleide klasse enkel aandacht hoeft te hebben voor de extra attributen in de afgeleide klasse en erop dient te vertrouwen dat de destructor van de basisklasse correct werkt! Volgend code-voorbeeld toont drie klassen A, B en C, die hiërarchisch van elkaar overerven:

```

class A {
public:
    ~A();
    ...
private:
    ...

```

```

};

class B : public A {
public:
    ~B();
...
private:
...
};

class C : public B {
public:
    ~C();
...
private:
...
};

```

Wanneer een object de klasse C out-of-scope gaat (zoals hieronder getoond) wordt eerst de destructor van C opgeroepen, vervolgens de destructor van B, en tenslotte de destructor van A.

```

{
C c;
...

} // c out of scope
// ~C wordt opgeroepen, ~B wordt opgeroepen, ~A wordt opgeroepen,
// (in deze volgorde!)

```

De destructoren worden dus in omgekeerde volgorde van de constructoren aangeroepen.

13.2.6 Toekenning-operator =

Beschouw volgende declaratie van een basisklasse A en afgeleide klasse B, waarbij beiden een toekenning-operator (Eng.: assignment operator) hebben:

```

class A {
public:
    A();
    A(int i);
    void setVarA(int i);
    int getVarA();
    A& operator=(const A& a);
private:
    int varA;
};

class B : public A {
public:

```

```

B();
B(int i);
B(int varA, int varB);
void setVarB(int i);
int getVarB();
B& operator=(const B& b);
private:
    int varB;
};

```

De implementatie van deze operator in de afgeleide klasse B wordt hieronder weergegeven:

```

B& B::operator=(const B& b){
    A::operator=(b);
    varB=b.varB;
    return *this;
}

```

Er wordt dus eerst de toekenning-operator van de basisklasse opgeroepen en vervolgens vindt de toekenning aan de attributen van de afgeleide klasse plaats.

13.2.7 protected en private overerving

Naast publieke overerving zijn ook **protected** en **private** overerving mogelijk. Deze worden echter zelden gebruikt. Beschouw de volgende basisklasse A:

```

class A {
public:
    ~A();
    ...
private:
    ...
};

```

Volgende declaratie illustreert **protected** overerving bij de klasse B:

```

class B : protected A {
public:
    ...
private:
    ...
};

```

Resultaat is dat publieke methoden en attributen uit de basisklasse A in de afgeleide klasse B nu **protected** worden. Volgende declaratie toont **private** overerving bij de klasse C:

```

class C : private A {
public:
    ...
};

```

```

private:
...
};
```

Resultaat is dat alle attributen en methoden uit de basisklasse A in de afgeleide klasse C nu **private** worden. De spaties voor en na : mogen opnieuw, maar hoeven niet (zowel bij **protected** als **private** overerving).

13.2.8 Meervoudige overerving

Een afgeleide klasse in C++ kan meerdere basisklassen hebben (in tegenstelling tot Java). Onderstaand voorbeeld toont de declaratie van de klasse C, die overerft van zowel klasse A als klasse B:

```

class A {
public:
    ~A();
    ...
private:
    ...
};

class B {
public:
    ...
private:
    ...
};

class C : public A, public B {
public:
    ...
private:
    ...
};
```

Bij meervoudige overerving dient wel sterk opgelet te worden voor mogelijke dubbelzinnigheden: als klassen A en B ook een gemeenschappelijk basisklasse zouden hebben, zijn de attributen van deze gemeenschappelijke basisklasse tweemaal aanwezig in de klasse C en is het voor de compiler niet duidelijk welk attribuut bedoeld wordt (tenzij het expliciet met :: aangegeven wordt). Ook indien klasse A en klasse B dezelfde methode uit de gemeenschappelijke basisklasse hebben geherdefinieerd, en deze is niet geherdefinieerd in klasse C, dan is het bij het oproepen van deze methode op een object van de klasse C niet duidelijk welke van de twee methoden dient gekozen te worden (tenzij dit opnieuw expliciet met :: aangegeven wordt).

13.3 Polymorfisme

13.3.1 Concept virtuele functies

Beschouw volgende declaratie van een basisklasse **Figure**, die twee private attributen bijhoudt: de coördinaten van het middelpunt van de figuur. Er zijn twee methoden: **draw()** om de figuur te tekenen en **drawCenter()** om de figuur in het midden van een scherm te tekenen.

```
class Figure{
public:
    void draw();
    void drawCenter();
private:
    int x;
    int y;
};
```

Volgende twee klassen erven over van de basisklasse **Figure**:

```
class Circle: public Figure{
public:
    void draw();
private:
    double radius;
};

class Rectangle: public Figure{
public:
    void draw();
private:
    int height;
    int width;
};
```

De klasse **Circle** en **Rectangle** hebben beiden private attributen om respectievelijk een cirkel en rechthoek te kunnen voorstellen ten opzichte van de coördinaten van het middelpunt in de basisklasse. Bovendien hebben ze beiden een **draw()** methode omdat elke klasse zijn eigen logica heeft om een figuur te kunnen tekenen.

De methode **drawCenter()** is enkel aanwezig in de basisklasse **Figure**. De implementatie van deze basisklasse is als volgt:

```
void Figure::drawCenter() {
    // bepaal centrum
    draw(); // teken in centrum
}
```

Beschouwen we nu het aanmaken van objecten van de afgeleide klassen en het oproepen van deze **drawCenter()** methode:

```

Rectangle r;
Circle c;

r.drawCenter();
c.drawCenter();

```

Het gewenste gedrag bij beide bovenstaande methode-oproepen zou zijn dat de methode `drawCenter()` de methode `draw()` van de afgeleide klasse oproept: enkel in de afgeleide klasse is de logica aanwezig om een figuur van een specifiek type op een correcte manier te tekenen. In tegenstelling tot Java, is het echter in C++ zo dat de `draw()` methode uit de basisklasse `Figure` opgeroepen wordt! Reden hiervoor is dat `drawCenter()` een methode uit de basisklasse is en deze enkel methoden tot op dit niveau kan oproepen. Wanneer een nieuwe afgeleide klasse aangemaakt wordt, bijvoorbeeld de klasse `Triangle` (die een driehoek voorstelt):

```

class Triangle: public Figure{
public:
    void draw();
private:
    //3 endpoint coordinates
};

Triangle r;
r.drawCenter();

```

dan zou de methode `drawCenter()` uit de basisklasse de `draw()` methode uit de onderliggende klasse `Triangle` dienen op te roepen. Probleem hiermee is echter dat de nieuwe klasse `Triangle` nog niet bestond toen de methode `drawCenter()` in de basisklasse werd geïmplementeerd!

De oplossing hiervoor in C++ is het gebruik van *virtuele functies*. De essentie van virtuele functies is dat ze kunnen gebruikt worden vooraleer ze gedefinieerd zijn. De koppeling met de specifieke functie-code gebeurt bij gebruik (afhankelijk van het type van het object, waarop de methode opgeroepen wordt), en niet bij de implementatie. Een virtuele functie wordt gedeclareerd door het `virtual` sleutelwoord te vermelden voor de methode declaratie. Hieronder wordt de declaratie van de klassen `Figure`, `Circle`, `Rectangle`, en `Triangle` getoond met virtuele functies:

```

class Figure{
public:
    virtual void draw();
    void drawCenter();
private:
    int x;
    int y;
};

class Circle: public Figure{
public:

```

```

    virtual void draw();
private:
    double radius;
};

class Rectangle: public Figure{
public:
    virtual void draw();
private:
    int height;
    int width;
};

class Triangle: public Figure{
public:
    virtual void draw();
private:
    //3 endpoint coordinates
};

```

Indien nu de volgende code uitgevoerd wordt:

```

Rectangle r;
Circle c;
Triangle r;

r.drawCenter();
c.drawCenter();
r.drawCenter();

```

dan worden telkens de correcte `draw()` methoden opgeroepen.

Bemerkt dat het `virtual` sleutelwoord herhaald wordt bij de methode-declaraties in de afgeleide klassen: dit is niet strikt nodig (vermelding van `virtual` bij de methode in de basisklasse volstaat), maar is wel aangeraden (omdat een blik van een ontwikkelaar op de declaratie van de afgeleide klasse dan volstaat om te zien welke methoden `virtual` zijn en welke niet, de declaraties van de basisklasse en de afgeleide klasse hoeven dan niet met elkaar vergeleken te worden om te zien welke methoden `virtual` zijn en welke niet).

Het gebruik van virtuele functies is een zeer belangrijk principe bij object-georiënteerd programmeren. In Java zijn *alle* functies virtueel en wordt het `virtual` sleutelwoord bijgevolg niet gebruikt.

De compiler creëert een tabel met wijzers naar de virtuele functies (dit wordt de *virtuele functie tabel* genoemd), elke pointer wijst naar de locatie van de code voor die functie. Bij het oproepen van een methode op een object wordt de correcte wijzer geselecteerd en wordt de gepaste functie-code uitgevoerd. Dit principe wordt *late binding* of *dynamische binding* genoemd: de correcte code van een methode oproep wordt bij het oproepen zelf bepaald.

In C++ is het dus de verantwoordelijkheid van de programmeur om te beslissen welke

methoden virtueel zijn en welke niet. Het virtueel maken van een methode zorgt er enerzijds voor dat een virtuele functie tabel aangemaakt dient te worden (extra geheugen) en dat anderzijds bij het uitvoeren de juiste wijzer dient geselecteerd te worden (extra controle die een iets tragere uitvoering tot gevolg heeft). Gebruik van virtuele methoden zorgt dus wel voor overhead bij de uitvoering. In Java zijn alle methoden virtueel, C++ laat wel toe dat de programmeur kan kiezen of hij de overhead al dan niet nodig vindt. De implementatie van een virtuele functie in een afgeleide klasse wordt overschrijving (Eng.: overriding) genoemd, in plaats van herdefinitie (zoals in sectie 13.2.3 aan bod kwam). Herdefinitie en overschrijving komen op hetzelfde neer, maar overschrijving is de terminologie die steeds voor virtuele functies gehanteerd wordt.

13.3.2 Abstracte basisklassen

Een basisklasse kan soms geen betekenisvolle implementatie geven voor sommige methoden, dit kan bijvoorbeeld enkel zinvol in de afgeleide klassen. In de basisklasse **Figure** hierboven kan aan de `draw()`-methode geen betekenisvolle implementatie gegeven worden. De oplossing hiervoor is om van deze methode een *pure virtuele methode* te maken, met als syntax:

```
virtual void draw() = 0;
```

De toevoeging `= 0` geeft aan dat het om een pure virtuele functie gaat: deze vereisen geen definitie in de klasse zelf, maar wel in de afgeleide klassen, het verplicht de afgeleide klassen om een implementatie te voorzien.

Een klasse met één of meerdere pure virtuele functies, wordt een abstracte basisklasse genoemd: het kan enkel als basisklasse gebruikt worden en er kunnen geen objecten van deze klasse geïnstantieerd worden. Indien in een afgeleide klasse niet alle pure virtuele functies gedefinieerd worden, wordt deze afgeleide klasse ook een abstracte basisklasse. Het gebruik van een pure virtuele functie in C++ komt dus neer op het gebruik van het **abstract** sleutelwoord in Java. Het sleutelwoord **abstract** bestaat niet in C++.

Voor de volledigheid wordt hieronder de volledige declaratie van de basisklasse **Figure** en de afgeleide klassen getoond:

```
class Figure{
public:
    virtual void draw()=0;
    void drawCenter();
private:
    int x;
    int y;
};

class Circle: public Figure{
public:
    virtual void draw();
private:
    double radius;
```

```

};

class Rectangle: public Figure{
public:
    virtual void draw();
private:
    int height;
    int width;
};

class Triangle: public Figure{
public:
    virtual void draw();
private:
    //3 endpoint coordinates
};

```

13.3.3 Toekenning afgeleide klassen

Met de toekenning (operator `=`) van objecten van basisklassen aan objecten van afgeleide klassen, dient omzichtig omsprongen te worden. Beschouw de basisklasse A en de afgeleide klasse B:

```

class A {
public:
    void setVarA (int i);
    int getVarA ();
private:
    int varA;
};

class B : public A {
public:
    void setVarB (int i);
    int getVarB ();
private:
    int varB;
};

```

Volgende code illustreert de toekenning van objecten van de klasse A en B aan elkaar:

```

A a;
B b;
a=b; // OK
b=a; // niet OK !

```

De eerste toekenning werkt perfect: een object van een afgeleide klasse kan steeds toegekend worden aan een object van een basisklasse. In de tweede regel wordt de toekenning in de andere richting geprobeerd, hetgeen door de compiler niet toegestaan wordt en

resulteert in een foutmelding. Volgende code illustreert dat bij de toekenning `a=b` het object `a` enkel de gegevens van de basisklasse bevat:

```
A a;
B b;
b.setVarA(12);
b.setVarB(7);
a=b; // OK
cout << a.getVarA() << endl; //OK
cout << a.getVarB() << endl; // niet OK! compilatiefout
```

In sommige gevallen is het echter wel sterk aangewezen dat het object `a` na toekenning nog gegevens van de afgeleide klasse heeft: dit kan enkel indien pointers naar objecten gebruikt worden (zoals uitgelegd in de volgende sectie).

13.3.4 Toekenning pointers naar klassen

Beschouw opnieuw de klasse declaraties van `Figure` en `Rectangle`:

```
class Figure{
public:
    virtual void draw();
    void drawCenter();
private:
    int x;
    int y;
};

class Rectangle: public Figure{
public:
    virtual void draw();
private:
    int height;
    int width;
};
```

In volgende code wordt een wijzer naar een object van de basisklasse en een wijzer naar een object van de afgeleide klasse aangemaakt (`fp` staat voor figure pointer and `rp` staat voor rectangle pointer) en vervolgens wordt via `new` een object van de afgeleide klasse `Rectangle` aangemaakt:

```
Figure* fp1;
Rectangle* rp1;
rp1=new Rectangle;
fp1=rp1;
fp1->draw(); //roept draw van Rectangle op!
```

De toekenning van de pointers lukt perfect en via de pointer `fp1` kunnen nog perfect de virtuele methoden van de afgeleide klassen opgeroepen worden. Er kan ook rechtstreeks met pointers naar de basisklasse alleen gewerkt worden, zoals hieronder geïllustreerd:

```
Figure* fp2 = new Rectangle;
fp2->draw(); //roept draw van Rectangle op!
```

13.3.5 Datastructuren met polymorfe objecten

Het is zéér handig om in een datastructuur verschillende objecten op te slaan, die als gemeenschappelijk kenmerk hebben dat ze overgeerfd zijn van eenzelfde basisklasse, bijvoorbeeld een **array** van figuren (basisklasse **Figure**): deze array bevat dan objecten, die ofwel rechthoeken (afgeleide klasse **Rectangle**), cirkels (afgeleide klasse **Circle**) of driehoeken (afgeleide klasse **Triangle**) kunnen zijn. Belangrijk bij datastructuren is dat ze homogeen dienen te zijn: alle elementen dienen van hetzelfde type te zijn, hetgeen impliceert dat bij opslag van objecten van verschillende afgeleide klassen als type van de elementen van de array het type van de basisklasse dient gekozen te worden. Vermits toekenning van objecten van een afgeleide klasse aan objecten van een basisklasse voor problemen zorgt (zoals hierboven in sectie 13.3.3 uitgelegd), is de **enige** mogelijkheid in dit geval het kiezen van een pointer-type voor het type van de elementen in de datastructuur. In het voorbeeld van een array van figuren, dient men dus de volgende declaratie te nemen om een array van objecten van de klasse **Figure** aan te maken:

```
Figure** figureArray;
```

en het creëeren, opvullen, gebruiken en verwijderen van de array gaat dan bijvoorbeeld als volgt:

```
figureArray = new Figure* [4];
figureArray[0] = new Circle;
figureArray[1] = new Rectangle;
figureArray[2] = new Circle;
figureArray[3] = new Triangle;
...
for(int i=0;i<4;i++)
    figureArray[i]->draw();
...
delete[] figureArray;
```

Men noemt objecten in een dergelijke datastructuur **polymorf**: ze kunnen verschillende vormen aannemen, maar hebben wel als eigenschap over te erven van een gemeenschappelijke basisklasse. De noodzakelijkheid om hiervoor pointers naar de basisklasse (Eng.: base pointers) te gebruiken, is een zéér belangrijk principe in C++.

13.3.6 Casting

Casting betekent de conversie van een type naar een ander type, in dit geval de conversie van een object van een basisklasse naar een object van een afgeleide klasse en omgekeerd. De regel is dat een object van een basisklasse niet naar een object van een afgeleide klasse kan gecast worden, maar omgekeerd wel. Volgende code illustreert een poging om een object van een basisklasse om te zetten naar een object van een afgeleide klasse (via **static_cast**, zoals uitgelegd in hoofdstuk 10).

```
Figure f;
Circle c;
...
c = static_cast<Circle>(f); //niet toegelaten
```

Dit wordt *downcasting* genoemd. In omgekeerde volgorde lukt dit wel perfect, zoals hieronder geïllustreerd:

```
f = c; // toegelaten
f = static_cast<Figure>(c); //eveneens toegelaten
```

Dit wordt *upcasting* genoemd, hetgeen geen enkel probleem vormt. Downcasting is soms wel vereist, zeker indien pointers naar objecten gebruikt worden, zoals in sectie 13.3.4 aan bod kwam. Dit wordt dan uitgevoerd door middel van `dynamic_cast`, zoals hieronder geïllustreerd wordt:

```
Figure *fp;
fp = new Circle;
Circle *cp = dynamic_cast<Circle*>(fp);
if (cp) { //gelukt}
else { //mislukt }
```

De return waarde van `dynamic_cast` is het omgezette object of `NULL` indien de omzetting niet gelukt is. Controle van de return waarde (zoals in het voorbeeld hierboven) laat dus toe om te checken of de omzetting gelukt is. Casting naar een afgeleid type lukt dus als het gaat om pointers naar een object van de basisklasse, die wijzen naar een object van de afgeleide klasse. **Belangrijk** bij downcasting is dat het enkel werkt als er minstens één virtuele functie aanwezig is (de virtuele functietabel wordt namelijk gebruikt bij uitvoering van `dynamic_cast` en deze is enkel aanwezig als er één of meerdere virtuele functies in de declaratie van de klasse aanwezig zijn).

13.3.7 Virtuele destructoren

Beschouw volgende code, waarin een pointer naar een object van een basisklasse wijst naar een object van de afgeleide klasse (aangemaakt met `new`):

```
Figure *fp = new Rectangle;
...
delete fp;
```

De `delete` opdracht zorgt enkel voor het aanroepen van de destructor van de basisklasse en **niet** van de destructor van de afgeleide klassen. De oplossing hiervoor is om de destructor virtueel te maken (d.w.z. de declaratie vooraf te laten gaan door het `virtual` sleutelwoord). Het wordt aanzien als een goede methodologie om *alle* destructoren virtueel te declareren wanneer men werkt met afgeleide klassen in combinatie met pointers.

Hoofdstuk 14

Templates

14.1 Functie templates

14.1.1 Introductie

Beschouw volgende functie definitie, waarbij de waarde van de twee gehele argumenten omgewisseld wordt:

```
void wissel(int& var1, int& var2){  
    int temp;  
    temp = var1;  
    var1 = var2;  
    var2 = temp;  
}
```

Deze functie kan enkel gebruikt worden indien argumenten van het type `int` doorgegeven worden, niet voor variabelen van een ander type (bijv. `double`) of zelf-gedefinieerde objecten (i.e. instanties van zelf-gedefinieerde klassen). Het zou zeer handig zijn om een functie te kunnen declareren en definiëren die voor alle type objecten werkt.

Dankzij functie overloading in C++ (hoofdstuk 12) kan de naam van een functie hergebruikt worden, maar de argument-types dienen telkens aangepast te worden (zowel in de declaratie als in de implementatie) als men nieuwe types als argument aan deze functie wil meegeven. Daarom is het meer aangewezen om gebruik te maken van een belangrijk concept in C++, namelijk templates (Ned.: sjablonen). Dit wordt uitgelegd in de verdere secties van dit hoofdstuk.

14.1.2 Functie template syntax

Een functie template wordt opgebouwd door het volgende toe te voegen aan een functie definitie:

1. de prefix `template<class T>`: geeft aan dat het om een functie template gaat en dat `T` als type parameter fungert. Bij gebruik van de functie wordt dan telkens het correcte type gesubstitueerd in `T`.

2. de type parameter `T` wordt in de implementatie gebruikt. `T` wordt meestal als naam gebruikt, maar er mag ook een andere naam gekozen worden.

In plaats van het sleutelwoord `class` in de prefix, mag ook het sleutelwoord `typename` gebruikt worden: beiden zijn volledig equivalent. Meestal wordt echter `class` gebruikt. Bemerk dat ook primitieve types (bijv. `int`) als type parameter kunnen gebruikt worden. Volgende code illustreert de functie `wissel` van hierboven als functie template:

```
template<class T>
void wissel (T& var1, T& var2){
    T temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

Deze functie kan dan bijvoorbeeld op de volgende manier gebruikt worden:

```
int i1=2,i2=4;
double d1=3.0, d2=5.3;
A a1(3), a2(6);

wissel<int>(i1,i2);
wissel<double>(d1,d2);
wissel<A>(a1,a2);
```

De grote kracht van functie templates is dus dat één implementatie met een type parameter volstaat om voor verschillende types te kunnen gebruikt worden (met telkens tussen `< >` (scherpe haakjes) het eigenlijke type).

14.1.3 Meerdere types als parameter

Het gebruik van templates is niet beperkt tot één type parameter: er kunnen ook meerdere parameters gebruikt worden. Volgende code illustreert de declaratie van een functie template met twee parameters `T1` en `T2`:

```
template<class T1, class T2>
T1 functie_naam (const T1& t1, const T2& t2);
```

Meestal wordt er echter slechts één type parameter gebruikt. De compiler controleert telkens of alle types wel gebruikt worden in de implementatie.

Volgende declaratie is bijvoorbeeld ook toegelaten:

```
template<class T>
T functie_naam (int i, const T& t1, const T& t2, double d);
```

Dit voorbeeld illustreert dat ook argumenten met concrete types (zonder template parameters) kunnen gebruikt worden (in dit geval het eerste en vierde argument).

14.1.4 Aanbevolen aanpak

Volgende stappen worden best doorlopen bij implementatie van functie templates:

1. ontwikkel de functie met een specifiek type (bijv. `int` of `std::string`),
2. debug deze functie grondig,
3. converteer vervolgens de functie naar een functie template (i.e. vervang de type namen door de type parameter).

Het voordeel van deze aanpak is dat grondige debugging kan uitgevoerd worden in stap 2. en dat de nadruk bij de implementatie ligt op het algoritme zelf en niet op de syntax bij het gebruik van templates.

14.1.5 Toegelaten parameter-types in templates

Zoals kan aangevoeld worden is er een beperking op de klassen die in de template parameters kunnen gesubstitueerd worden. Enkel klassen met volgende eigenschappen mogen in `T` gesubstitueerd worden :

1. de klasse beschikt over een geldige assignment operator (`=`) en copy constructor,
2. de klasse beschikt over een correcte destructor (zeker belangrijk wanneer er dynamisch gealloceerd geheugen gebruikt wordt).

Bovendien kunnen er in de klasse voorzieningen vereist zijn voor alle operatoren en functies die in de template implementatie op de objecten van deze klasse opgeroepen worden, bijvoorbeeld een `+` operator of een `*` (vermenigvuldiging) operator.

Volgende code werkt niet correct:

```
int a[10], b[10];
wissel(a, b);
```

Reden hiervoor is dat de toekenning (`=`) niet werkt voor array's (cfr. sectie 7.2.11): `a` en `b` in dit voorbeeld bevatten een constant adres, dat niet kan aangepast worden!

14.2 Klasse templates

14.2.1 Overzicht

Naast functie templates (vorige sectie), kunnen ook templates van klassen aangemaakt worden. De type parameter geeft dan meestal het type van de attributen in de klasse weer. Vermits de member methoden dit type ook dikwijls als één van de argumenten hebben, zijn de member methoden van een klasse template ook automatisch functie templates.

Beschouw volgende declaratie van de klasse template `Pair`:

```
template<class T>
class Pair{
public:
    Pair();
    Pair(T firstVal, T secondVal);
    void setFirst(T newVal);
    void setSecond(T newVal);
    T getFirst() const;
    T getSecond() const;
private:
    T first;
    T second;
};
```

Deze klasse bevat twee attributen, beide van het type T en in de signaturen van de methoden komt telkens T voor (behalve in de default constructor `Pair()`, welke geen argumenten heeft).

Voor de implementatie van de member methoden dient men rekening te houden met het feit dat elke methode een functie template is en dus de template prefix telkens vereist is. Vermits de methoden ook onderdeel van de klasse zijn, dienen ze vooraf gegaan te worden door `klasse_naam<T>::`. De naam van de constructor is `klasse_naam` en de naam van de destructor `~klasse_naam`.

Volgende code illustreert de implementatie van de constructor en de methode `setFirst`:

```
template<class T>
Pair<T>::Pair(T firstVal, T secondVal){
    first = firstVal;
    second = secondVal;
}

template<class T> void Pair<T>::setFirst(T newVal){
    first = newVal;
}
```

Eens de klasse template gedeclareerd en geïmplementeerd is, kan deze worden gebruikt voor het aanmaken van objecten. Volgend code-voorbeeld illustreert dit:

```
Pair<int> pair1;
Pair<char> pair2;
pair1.setFirst(3);
pair2.setSecond('0');
```

14.2.2 Klasse templates binnen functie templates

Klasse templates kunnen ook als argument of type van de return waarde van niet-member methoden gebruikt worden. Volgend voorbeeld toont een functie template `sum<T>` die de klasse template `Pair<T>` als argument neemt. Dezelfde type parameter T komt voor in zowel de functie template als de klasse template. De functie berekent de som van de twee waarden in het argument:

```
template<class T> T sum(const Pair<T>& pair);
```

Een belangrijke voorwaarde voor de te substitueren types in T in dit voorbeeld is dat de operator + dient gedefinieerd te zijn voor waarden van type T.

14.2.3 Aanbevolen aanpak voor ontwerp van klasse templates

De aanbevolen aanpak bij het ontwerpen van klasse templates is identiek als bij functie templates:

1. declareer en implementeer de klasse met een specifiek type (bijv. int of std::string),
2. debug deze klasse grondig, door alle methoden op te roepen en te testen,
3. converteer vervolgens de klasse naar een klasse template (i.e. vervang de type namen door de type parameter of type parameters).

14.3 Instantiatie en compilatie van templates

De compiler genereert de definities van enerzijds (i) alle gebruikte functie templates en anderzijds (ii) alle klasse template methoden wanneer klasse templates gebruikt worden (door substitutie van de types tussen < >). Deze generatie wordt template instantiatie genoemd. Vervolgens wordt de gegenereerde code dan gecompileerd.

Normaalgezien worden templates geïmplementeerd in een headerbestand, omdat voor de compiler zowel definitie als implementatie zichtbaar moeten zijn op het moment van instantiëring van een template en omdat functie en klasse templates doorgaans klein in omvang zijn.

Indien de declaratie en implementatie opgesplitst wordt in headerbestanden en cpp-bestanden, dient een extra bestand (met bijvoorbeeld als naam `template_instantiations.cpp`) toegevoegd te worden: dit bestand bevat de declaratie van de templates met ingevulde type parameters en zal de compiler迫eren om de code te genereren voor de functie en/of klasse templates met deze type parameters.

Beschouw volgend voorbeeld van een `template_instantiations.cpp` bestand:

```
#include "BinaryTree.cpp"
#include <string>

template class BinaryTree<std::string, int>;
```

Dit bestand zorgt ervoor dat de code gegenereerd wordt voor de `BinaryTree` klasse template met typeparameters `string` (i.e. type voor de sleutels in de binaire zoekboom) en `int` (i.e. type voor de waarden in de binaire zoekboom). Dit bestand hoeft nergens geïncludeerd te worden en dient enkel om de compiler te verplichten de juiste code te genereren.

Indien een dergelijk bestand niet toegevoegd wordt, resulteert dit in linkerfouten.

14.4 Template type definities

Via `typedef` kan een volwaardig nieuw type gedeclareerd worden, dat kan gebruikt worden voor verdere object-declaraties. Beschouw bijvoorbeeld de type definitie van `PairOfInt`:

```
typedef Pair<int> PairOfInt;
```

en het aanmaken van de objecten `pair1` en `pair2` van dit type:

```
PairOfInt pair1, pair2;
```

Via `typedef` kan dus vermeden worden dat `< >` dient gebruikt te worden bij declaratie van objecten.

14.5 Templates en overerving

Klasse templates kunnen overerven van andere klassen: de basisklassen kunnen klasse templates zijn met dezelfde type parameter(s), maar dit hoeft niet: overerven van klassen zonder template parameters is evengoed mogelijk. De afgeleide klasse van een klasse template is ook een klasse template.

De syntax voor afleiding is dezelfde als deze bij gewone afleiding.

14.6 Oefening

Ontwerp een klasse template voor een binaire zoekboom: `BinaryTree` met één type parameter `T`, die het type van de elementen voorstelt, die in de binaire zoekboom opgeslagen worden. Veronderstel dat de klassen die in `T` gesubstitueerd worden een geldige `<` operator hebben (de klasse `std::string` voldoet aan deze voorwaarde). Maak onderscheid tussen twee gevallen:

1. de nodes van de binaire zoekboom bevatten de elementen zelf (diepe kopieën),
2. de nodes van de binaire zoekboom bevatten niet de elementen zelf, maar een wijzer (Eng.: pointer) naar het corresponderende element (belangrijk in kader van opslag van polymorfe objecten, cfr. hoofdstuk 13, overerving en polymorfisme).

Hoofdstuk 15

Standard Template Library (STL)

15.1 Inleiding

STL (Eng.: Standard Template Library) werd in 1994 ontwikkeld als de standaard bibliotheek voor C++: het bevat datatypes, algoritmen en hulpklassen. De belangrijkste onderdelen zijn:

1. containers: dit zijn klassen, die dienen voor dataopslag en gebaseerd zijn op een bepaalde datastructuur. Er zijn zeven container-types mogelijk in STL: `vector`, `deque`, `list`, `set`, `multiset`, `map` en `multimap`. Deze komen in dit hoofdstuk allen aan bod.
2. iteratoren: dit zijn klassen die dienen om efficiënt door alle aanwezige elementen in de containers te itereren.
3. algoritmen: dit zijn functies die een bepaald algoritme implementeren, bijvoorbeeld een sorteer-, zoek- of filteralgoritme.

Deze drie onderdelen worden in dit hoofdstuk uitgebreid behandeld.

STL is volledig gebaseerd op templates: de containers zijn allen klasse templates. De achterliggende code is nauwkeurig ontwikkeld om zo snel mogelijke uitvoering toe te laten.

15.2 Voorbeeld: `vector` container

Eén van de zeven containers is de `vector` container: deze laat toe om elementen achteraan toe te voegen en te verwijderen. De toegevoegde elementen worden in volgorde bewaard.

15.2.1 Code voorbeeld

Beschouw volgende code, waarin een `vector` van `int`'s aangemaakt wordt, de getallen ingelezen en opgeslagen worden in de `vector` tot wanneer een niet-numeriek karakter

ingegeven wordt. Vervolgens worden de elementen van de vector gesorteerd en worden de elementen in volgorde afgedrukt:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void main(){
    vector<int> v;
    int input;
    while(cin >> input)
        v.push_back(input);
    sort(v.begin(), v.end());
    int n = v.size();
    for(int i = 0; i < n; i++){
        cout << v[i] << endl;
    }
}
```

Laten we deze code nu stap voor stap beschouwen:

```
#include <vector>
#include <algorithm>
```

Includeren van deze headerbestanden is noodzakelijk om het **vector** type uit STL te kunnen gebruiken en ook de algoritmen te kunnen oproepen (in dit geval het sorteeralgoritme aan de hand van de functie **sort**).

```
vector<int> v;
```

Maakt een vector **v** aan: alle STL containers (inclusief **vector**) zijn klasse templates, bij gebruik dient het type van de elementen tussen scherpe haakjes (**< >**) gespecificeerd te worden. In dit voorbeeld wordt een **vector** voor de opslag van gehele getallen aangemaakt.

```
v.push_back(input);
```

De methode **push_back** zorgt voor het opslaan van een nieuw element *achteraan* in de **vector**. Een **vector** implementeert een LIFO (Eng.: Last In, First Out) datastructuur, ook wel stapel (Eng.: stack) genoemd.

```
sort(v.begin(), v.end());
```

De functie **sort** implementeert een sorteeralgoritme, dat de elementen in de **vector** van klein naar groot plaatst. Als argumenten worden twee *iteratoren* meegegeven, die het bereik van de elementen voor sortering aangeven: in dit geval wordt de volledige **vector** gesorteerd (van begin tot einde). Iteratoren komen verder in dit hoofdstuk aan bod (secties 15.2.4 en 15.7).

```
int n = v.size();
```

De **size** functie laat toe om de grootte van de vector (i.e. het aantal aanwezige elementen) op te vragen.

```
for(int i = 0; i < n; i++){
    cout << v[i] << endl;
}
```

De [] operator is eveneens gedefinieerd voor de klasse **vector**: het laat toe om rechtstreekse toegang te verkrijgen tot het element op positie **i** (posities beginnen bij 0 en lopen tot en met **size()-1**).

15.2.2 Constructoren

Volgende constructoren kunnen bijvoorbeeld gebruikt worden voor initialisatie van een **vector**:

```
vector<int> v;
// allocates 3 integers
vector<int> w(3);
// allocates 6 floats with value 3.1415
vector<float> u(6, 3.1415);
```

De eerste regel toont de initialisatie van een **vector** zonder het aantal elementen of de waarden te specifiëren. De tweede regel toont de initialisatie van een **vector** met plaats voor 3 elementen. De laatste regel toont de initialisatie van een **vector** met 6 elementen, die allen de waarde van het 2e argument krijgen.

15.2.3 Operaties op een vector

We beschouwen hieronder één voor één de operaties die op STL vectoren kunnen toegepast worden:

```
v.push_back(9);
v.pop_back();
```

push_back zorgt voor toevoegen van een element achteraan en **pop_back** voor het verwijderen van het laatste element in de **vector**.

```
v[3] = 7;
```

via de []-operator kan de waarde van elementen op een bepaalde positie aangepast worden.

```
int waarde = v.front();
waarde = v.back();
```

front() zorgt voor opvragen van het eerste element (wordt niet verwijderd), **back()** zorgt voor opvragen van het laatste element (wordt eveneens niet verwijderd).

```
int aantal = v.size();
```

`size()` zorgt voor opvragen van het aantal aanwezige elementen in de `vector`.

```
bool empty = v.empty();
```

De `empty()` methode laat toe om te controleren of de vector leeg is.

```
w = v;
```

De toekenning-operator (`=`) is eveneens geïmplementeerd voor vectoren, en zorgt voor individuele kopie van de elementen (de `=`-operator van de elementen wordt opgeroepen, en als deze laatste operator zorgt voor diepe kopie, worden de vectoren ook diep gekopieerd).

```
bool gelijk = (v == w);
```

De `==` operator laat toe om te controleren of de inhoud van twee vectoren identiek is: de `==` operator van de elementen in de vector wordt opgeroepen (indien deze correct geïmplementeerd is, worden de elementen perfect vergeleken).

```
w[2]--;
```

Via de `[]`-operator en de increment(`++`) en decrement(`--`) operatoren kunnen de elementen geïncrementeerd en gedecrementeerd worden.

```
v.swap(w);
```

Oproepen van `swap()` zorgt voor omwisselen van de inhoud van `v` en `w`.

15.2.4 `vector`: iteratoren

Iteratoren vervangen pointers in C. Een iterator bevat een wijzer naar een element in een container en laat toe om over alle elementen te itereren. Volgende methoden zijn hierbij belangrijk:

- `begin()`: wanneer toegepast op een container, geeft deze methode een iterator terug die wijst naar het eerste element.
- `end()`: wanneer toegepast op een container, geeft deze methode een iterator terug die wijst *voorbij* het laatste element. Via deze iterator waarde kan je dus geen geldig element uit de container opvragen.

Volgende code illustreert de aanmaak van twee iteratoren `i` en `j`, waarbij `i` geïnitialiseerd wordt via de `begin()` methode en `j` geïnitialiseerd wordt via de `end()` methode:

```
vector<int>::iterator i = v.begin();
vector<int>::iterator j = v.end();
```

Iteratoren kunnen bijvoorbeeld op volgende manier gebruikt worden om de aanwezige elementen van een container af te lopen:

```
vector<int>::iterator i = v.begin();
while (i != v.end()) {
    cout << *i << endl;
    i++;
}
```

De dereferentie-operator (`*`) zorgt voor het opvragen van de waarde op de plaats in de container, aangeduid door de iterator. De increment operator (`++`) zorgt voor het op-schuiven van de iterator naar het volgende element.

Iteratoren worden heel vaak gebruikt om een bereik (Eng.: range) aan te duiden: bijvoorbeeld (`v.begin()`, `v.end()`) geeft aan dat de volledige vector `v` beschouwd wordt, (`v.begin()`, `v.begin() + 4`) geeft aan dat de eerste 5 elementen van de vector `v` beschouwd worden. De `vector` iterator wordt een *random access iterator* genoemd, omdat via pointer arithmetiek de iteratoren willekeurig kunnen verhoogd en verlaagd worden. Volgend voorbeeld illustreert deze bewerkingen op iteratoren van de `vector` container.

```
i = i + n;
i -= n;
i[n];
```

De `vector` klasse beschikt ook over een constructor om de elementen van een andere vector te kopiëren bij constructie. Volgend voorbeeld illustreert de constructie van vector `w` op basis van de volledige vector `v`:

```
vector<int> w (v.begin( ), v.end( ));
```

Enkele operaties op vectoren werken ook met iteratoren als argument, bijvoorbeeld de `insert` en `erase` methoden zoals hieronder geïllustreerd (in commentaar wordt telkens de volgende regel uitgelegd):

```
vector<int> v(5, 9);
vector<int> w(4, 12);

// v : 9 9 9 9 9 -> v : 9 9 8 9 9 9
v.insert(v.begin() + 2, 8);

// insert v before w : 9 9 8 9 9 9 12 12 12 12
w.insert(w.begin( ), v.begin(), v.end( ));

// erase the last element
w.erase (w.end() - 1);

// erase everything beyond the third element
w.erase (w.begin() + 3, w.end());
```

15.2.5 Algoritmen

STL algoritmen werken ook steeds met iteratoren als argumenten. Beschouw volgend voorbeeld, waarbij een vector `v` gesorteerd wordt en een bepaald element in de gesorteerde vector gezocht wordt aan de hand van het binair zoekalgoritme.

```
/* sorts the indicated range (ascending) */
sort(v.begin(), v.end());
/* returns an iterator to the searched element, or end() */
binary_search(v.begin(), v.end(), x);
```

de `binary_search` methode veronderstelt een gesorteerde `vector` en past het binair zoeken algoritme (appendix 2) toe. De return waarde is een iterator, die wijst naar het gevonden element of het einde van de `vector` (`v.end()`) indien het element niet aanwezig was.

15.3 Functie objecten

Functie objecten (Eng.: function objects) zijn veralgemeende functiewijzers (Eng.: function pointers) en worden soms ook functoren (Eng.: functors) genoemd. Een belangrijke eigenschap is dat deze de operator () implementeren. Functie objecten worden door algoritmen als argument gebruikt om bijvoorbeeld elementen te vergelijken. Er zijn in STL standaard functie objecten gedefinieerd, voorbeelden hiervan zijn `greater<T>` en `less<T>` (die beiden toelaten om twee elementen te vergelijken en een `bool` als return waarde te geven).

Beschouw volgende declaratie en implementatie van het functie-object `odd`, waarbij de ()-operator overladen wordt:

```
class odd{
public:
    bool operator() (int a){
        return (a%2==1);
    }
};
```

Het gebruik van dit functie object `odd` wordt in volgend voorbeeld geïllustreerd:

```
void main(){
    vector<int> v(12,3);
    int n = 0;
    count_if(v.begin(), v.end(), odd, n);
    sort(v.begin(), v.end(), greater<int>);
}
```

de functie `count_if` neemt de functie object klasse `odd` als argument: het vierde argument `n` bevat na de oproep het aantal oneven elementen aanwezig in de `vector`. De oproep van de functie `sort` in dit voorbeeld sorteert de elementen van groot naar klein. Indien `less<int>` als derde argument zou meegegeven worden, worden de elementen van klein naar groot gesorteerd.

15.4 Types containers

In sectie 15.5 worden de containers `vector`, `deque` en `set` behandeld, dit zijn sequentiële containers vermits de elementen in volgorde van invoeging bewaard worden. In sectie 15.6 worden associatieve containers `set`, `multiset`, `map`, `multimap` behandeld.

Het type van de elementen `T` wordt tussen scherpe haakjes (`< >`) gespecificeerd bij het aanmaken van een container. De volgende methoden of operatoren voor het type `T` dienen aanwezig te zijn:

- publieke default constructor (zonder argumenten),
- publieke copy constructor,
- publieke destructor,
- publieke toekenning-operator (`=`).

Alle containers hebben dezelfde basis-interface en bieden allen de volgende methoden aan:

- `size`, `empty`
- `begin`, `end`
- `insert`, `erase`, `swap`
- `operator=`, `operator==`

Gebruik van deze methoden werd uitgelegd in kader van de `vector` container hierboven. Alle containers definiëren ook de types `iterator` en `const_iterator`.

Bemerkt dat de containers steeds homogeen zijn: enkel elementen van hetzelfde type kunnen opgeslagen worden en containers van verschillende element-types kunnen niet aan elkaar gelijkgesteld worden of gebruikt worden om elementen door te geven.

15.5 Sequentiële containers

15.5.1 `vector`

Een `vector` laat toe dat achteraan elementen toegevoegd en verwijderd worden. Volgende methoden zijn extra beschikbaar ten opzichte van de basisinterface:

- `push_back`, `pop_back`
- `front`, `back`
- `operator[]`

Gebruik van deze methoden werd uitgelegd aan de hand van voorbeelden hierboven.

15.5.2 deque

Een **deque** (Eng.: double ended queue) laat toe om elementen zowel vooraan als achteraan toe te voegen en te verwijderen. Volgende methoden zijn extra beschikbaar ten opzichte van de basisinterface:

- `push_back, pop_back`
- `front, back`
- `operator[]`
- `push_front, pop_front`

We verwijzen naar het **vector** voorbeeld voor uitleg bij deze methoden.

15.5.3 list

Een **list** laat toe dat elementen op willekeurige plaatsen kunnen toegevoegd worden. De implementatie is aan de hand van een dubbel gelinkte lijst. Volgende methoden zijn extra beschikbaar ten opzichte van de basisinterface:

- `push_back, pop_back`
- `front, back`
- `push_front, pop_front`
- geen `operator []!`
- methoden die een algoritme implementeren: `splice, remove, unique, merge, reverse, sort`, etc.

Net zoals bij de containers **vector** en **deque** verwijzen we naar het **vector** voorbeeld hierboven voor uitleg bij enkele van deze methoden.

15.6 Associatieve containers

15.6.1 Inleiding

Associatieve containers slaan elementen gesorteerd op volgens een sleutel (Eng.: key). De implementatie van deze containers is telkens gebaseerd op bomen, en meer bepaald worden rood-zwart bomen (Eng.: red black trees, appendix 2) gebruikt. Er zijn vier types van associatieve containers: **set**, **multiset**, **map** en **multimap**. In een **set** en **map** dienen alle sleutels uniek te zijn (geen *duplicate* sleutels). In een **multiset** en **multimap** kunnen dezelfde sleutels wel verschillende malen voorkomen.

Naast de basisinterface zijn ook volgende methoden beschikbaar:

- **find**: geeft een iterator terug naar het element dat gelijk is aan het argument in het opgegeven bereik,
- **count**: telt het aantal elementen met een waarde gelijk aan het argument in het opgegeven bereik,
- **lower_bound**: geeft een iterator terug naar het eerste element dat een sleutel heeft die groter is dan of gelijk aan het argument,
- **upper_bound**: geeft een iterator terug naar het eerste element dat een sleutel heeft die strikt groter is dan het argument.

15.6.2 set en multiset containers

Een **set** en **multiset** implementeren een verzameling van elementen, waarvan achteraf gemakkelijk kan nagegaan worden of ze al dan niet aanwezig zijn. Vermits de elementen gesorteerd opgeslagen worden, is het belangrijk dat de elementen kunnen vergeleken worden: daarom is de aanwezigheid vereist van de `<` operator in de klasse van elementen, of een `compare` functie, die twee elementen met elkaar kan vergelijken. Wanneer een standaard functie object gebruikt wordt, dient de klasse de `<` operator op de elementen te implementeren.

Beschouw bijvoorbeeld volgende declaratie van een **set**:

```
set<Employee, greater<Employee> >;
```

Het functie object `greater` maakt in dit geval gebruik van de `<` operator uit de klasse `Employee`:

```
bool greater(Employee x, Employee y) { return x > y; }
```

Eén vergelijkingsoperator (`<`) volstaat omdat de andere (`>` en `==`) hiermee kunnen geconstrueerd worden, bijvoorbeeld vergelijking van twee sleutels `k1` en `k2`

```
k1 == k2
```

kan worden geschreven als:

```
!(k1 < k2) && !(k2 < k1)
```

15.6.3 Hulpklasse template pair

STL definieert een hulpklasse template `pair<T1, T2>`. Deze laat toe om twee waarden van type `T1` en `T2`, respectievelijk te groeperen in één object. Deze klasse template beschikt over twee publieke attributen : `first` (eerste attribuut) en `second` (tweede attribuut).

In een `map` en `multimap` zijn elementen van het type `pair` opgeslagen (met `T1` het type van de sleutel en `T2` het type van de corresponderende waarde).

Beschouw bijvoorbeeld volgende invoeging van een element in een map `m`:

```
m.insert(pair<string, int>("hallo", 7));
```

Het resultaat van `insert` is een object van het type `pair` met twee elementen: een iterator die wijst naar het ingevoegde element en een attribuut van type `bool`, dat aangeeft of invoeging gelukt is.

Volgend code voorbeeld illustreert het invoegen van een element in een set `s` en het gebruik van de return waarde (van het type `pair`):

```
cout << *(s.insert(9).first);
if (s.insert(9).second)
    cout << "Insert completed";
else
    cout << "Insert failed";
```

Vermits in een `set` alle elementen uniek dienen te zijn en in dit voorbeeld hetzelfde element tweemaal wordt toegevoegd zal de toevoeging de tweede keer niet lukken (i.e. het tweede attribuut van de return waarde zal `false` zijn).

15.6.4 map container

De container `map` laat gebruik van de `[]`operator toe: deze wordt heel veel gebruikt. Beschouwen we volgend voorbeeld, waarbij een `map` aangemaakt wordt (met sleutels van het type `string` en gehele getallen als waarden):

```
map<string, int, less<string> > cnt;
pair<string, int> six("six", 6);
cnt.insert(six);
cnt["six"] = 5;
cnt["six"]++;
```

Zoals dit voorbeeld toont, geeft de `[]`operator met een sleutelwaarde als argument toegang tot de corresponderende waarde en laat het toe om de waarde aan te passen. Aanpassing van de sleutel is **niet** mogelijk.

15.7 Iteratoren

De iteratoren van de zeven STL containers zijn ofwel *random access iteratoren* ofwel *bidirectionele iteratoren*: op vector en deque zijn *random access iteratoren* mogelijk, op de andere (`list`, `set`, `multiset`, `map`, `multimap`) zijn enkel *bidirectionele iteratoren* mogelijk.

Random access iteratoren laten volgende operaties toe:

```
i--; i++; *i;
i=j; i == j; i != j; --i;
value = *i; *i = value;
i < j; i > j;
i + n; i - n;
i += n; i -= n; i[n];
```

Bidirectionele iteratoren laten enkel volgende operaties toe:

```
i--; i++; *i;
i=j; i == j; i != j; --i;
value = *i; *i = value;
```

Iteratoren worden altijd als onderdeel van de container klasse gedeclareerd, bijvoorbeeld:

```
vector<int>::iterator i;
list<Employee>::iterator j;
```

Soms is het handig om een iterator te gebruiken die *niet* toelaat om de elementen van de container aan te passen: de `const_iterator`, bijvoorbeeld:

```
vector<int>::const_iterator i = v.begin();
list<Employee>::const_iterator j = l.begin();
```

Belangrijk is dat iteratoren van associatieve containers **niet** kunnen gebruikt worden om de waarden van de sleutels te veranderen!

- `set` en `multiset`: laten enkel gebruik van `const_iterator` toe,
- `map` en `multimap`: enkel de waarde geassocieerd met een sleutel kan veranderen, **niet** de sleutel zelf.

Volgende code illustreert dit voor iteratie over een `map`:

```
map<string, int>::iterator i = m.begin();
(*i).second = 7; //aanpassing waarde: OK
(*i).first = "something"; //aanpassing sleutel: niet OK!
```

De compiler laat de laatste regel niet toe en stopt de compilatie. Een `const_iterator` wordt heel dikwijls in combinatie met een `map` of `multimap` container gebruikt.

15.8 Online referenties

Volgende online referenties zijn interessant als naslagwerk:

- Silicon Graphics: <http://www.sgi.com/tech/stl/>,
- Cppreference: <http://www.cppreference.com/cppstl.html>,

15.9 Uitgebreid code voorbeeld

In volgend voorbeeld worden polymorfisme en STL gecombineerd. Er wordt eerst een programma getoond, dat enkele fouten bevat. Nadien worden dan de verbeteringen besproken.

15.9.1 Originele versie

Beschouw volgende twee klassen, een basisklasse `Player` en een afgeleide klasse `Monster`:

```
/* bestand: player.h */
#include <iostream>
#include <string>

class Player
{
protected:
    std::string name;
public:
    Player(std::string setname) : name(setname) {}
    virtual void Display() {std::cout << "Hi, I'm " << name << ".";}
};

class Monster : public Player
{
public:
    Monster(std::string setname) : Player(setname) {}
    void Display() {std::cout << "I'm " << name << ", the monster!";}
};
```

In volgende main-functies worden deze klassen gebruikt:

```
/* bestand: main.cpp */
#include "player.h"
#include <list>
using namespace std;

main()
{
    list<Player> enemies;

    Player p1("Pete");
    Monster m1("Timmy");
    Monster m2("Gus");

    enemies.push_back(p1);
    enemies.push_back(m1);
    enemies.push_back(m2);

    list<Player>::iterator i;

    for(i=enemies.begin();i != enemies.end(); i++)
    {
        i->Display();
        cout << endl;
    }
}
```

Deze code levert de volgende output:

```
Hi, I'm Pete.  
Hi, I'm Timmy.  
Hi, I'm Gus.
```

Terwijl men de volgende verwacht:

```
Hi, I'm Pete.  
I'm Timmy, the monster!  
I'm Gus, the monster!
```

Vraag: wat is er fout in bovenstaande code?

In volgende sectie wordt de oplossing besproken.

15.9.2 Verbeterde versie

Vermits men polymorfisme wil benutten, dient men te werken met wijzers in de STL container.

```
/* bestand: main.cpp */  
#include "player.h"  
#include <list>  
using namespace std;  
  
main()  
{  
    list<Player*> enemies;  
  
    Player* p1 = new Player("Pete");  
    Monster* m1 = new Monster("Timmy");  
    Monster* m2 = new Monster("Gus");  
  
    enemies.push_back(p1);  
    enemies.push_back(m1);  
    enemies.push_back(m2);  
  
    list<Player*>::iterator i;  
  
    for(i=enemies.begin();i != enemies.end(); i++)  
    {  
        i->Display();  
        cout << endl;  
    }  
}
```

Deze aangepaste code levert echter een compilatiefout op de regel:

```
i->Display();
```

Deze dient aangepast te worden naar

```
(*i)->Display();
```

vermits de iterator nu een pointer bevat.

Het programma geeft nu de correcte output. Echter bevat het nog een geheugenlek. Vraag: waar bevindt zich het geheugenlek?

Voor de volledigheid wordt hier de volledig correcte code getoond:

```
/* bestand: main.cpp */
#include "player.h"
#include <list>
using namespace std;

main()
{
    list<Player*> enemies;

    Player* p1 = new Player("Pete");
    Monster* m1 = new Monster("Timmy");
    Monster* m2 = new Monster("Gus");

    enemies.push_back(pete);
    enemies.push_back(timmy);
    enemies.push_back(gus);

    list<Player*>::iterator i;

    for(i=enemies.begin();i != enemies.end(); i++)
    {
        (*i)->Display();
        cout << endl;
    }

    delete p1;
    delete m1;
    delete m2;
}
```

Hoofdstuk 16

Datastructuren in C++

Het is een uitstekende oefening om van volgende datastructuren een C++ klasse te ontwerpen: (voor de beschrijving van de datastructuren en de algoritmen voor de implementatie wordt voor de volledigheid verwezen naar Appendix 2).

16.1 Gelinkte lijsten

16.2 Boomstructuren

16.3 Hopen

16.4 Grafen

16.5 Hashtabellen

Deel IV

Software Ontwikkeling: Platformen en Technologieën

Hoofdstuk 17

Concurrent Version Systems (CVSs)

17.1 Definitie CVS

CVS is een server-gebaseerd systeem, dat bestanden centraal op een server opslaat. De server bevat alle data en wordt *repository* genoemd. Het CVS systeem laat toe dat:

- verschillende ontwikkelaars de bestanden kunnen downloaden,
- ontwikkelaars de bestanden simultaan (tegelijkertijd) kunnen aanpassen en duidelijk zicht hebben op elkaars aanpassingen,
- aanpassingen aan de bestanden bijgehouden worden (Eng: track changes) in versies van de bestanden,
- de verschillen tussen versies over de tijd eenvoudig kunnen nagekeken worden.

De benaming CVS (Eng.: Concurrent Version System) komt voort van het feit dat het systeem verschillende versies (Eng.: versions) van bestanden bijhoudt en dat gebruikers tegelijkertijd (Eng.: concurrent) aanpassingen aan de bestanden kunnen doorvoeren. CVS wordt soms verkeerdelyk als enkel een backup systeem gebruikt, zonder dat de bestanden in een deftige structuur georganiseerd zijn.

Een CVS systeem bevat steeds een gebruikersinterface (bijv. commandolijn op Unix/Linux of ingebet in Explorer of Shell op Windows of Mac). Via de gebruikersinterface zijn volgende basis-operaties in een CVS systeem mogelijk: `checkout`, `commit`, `update`. Deze worden in volgende sectie behandeld.

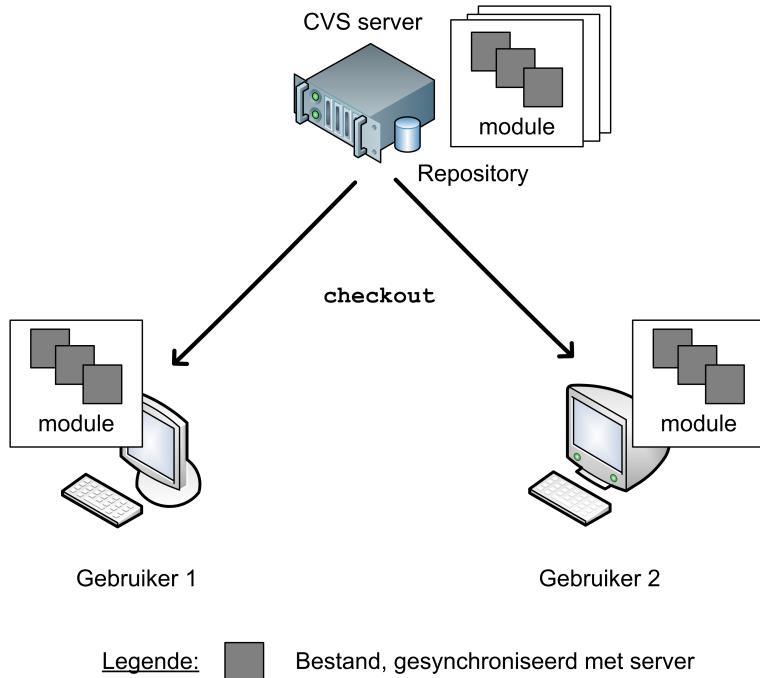
17.2 Operaties

17.2.1 `checkout`-operatie

Via de `checkout` operatie, wordt er een kopie van de bestanden van de server gedownload naar de lokale PC van de gebruiker (in een door de gebruiker gekozen folder). In principe kan iedereen met toegang tot de server alle bestanden downloaden, het is echter

mogelijk om op de server toegangsrechten in te stellen, zodat slechts een select publiek toegang heeft tot de bestanden.

Bestanden in de CVS *repository* zijn gegroepeerd in *modules*. Een module bevat bestanden, folders, etc. Figuur 17.1 illustreert het effect van de **checkout**-operatie in een CVS systeem.



Figuur 17.1: Principe van **checkout**-operatie in een CVS systeem.

17.2.2 commit-operatie

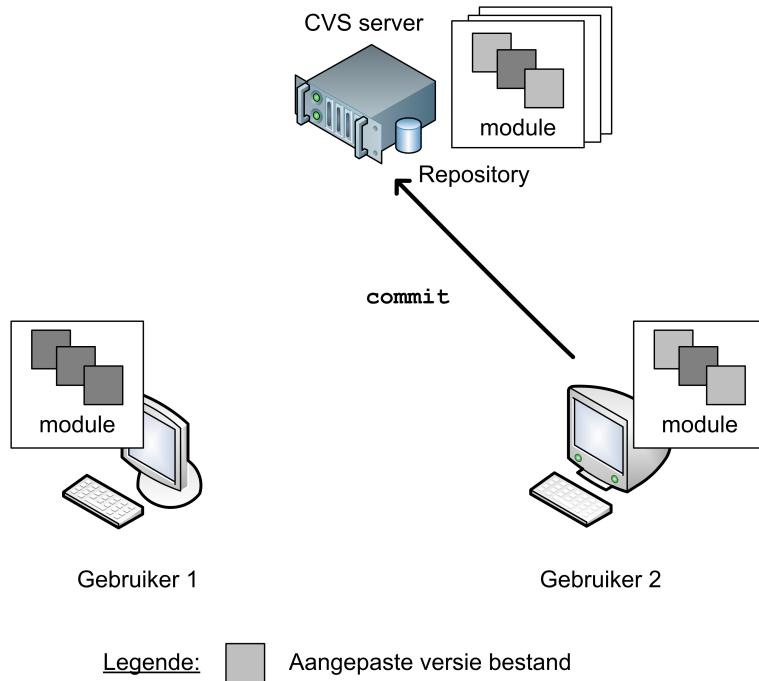
Na een **checkout**-operatie, kan een gebruiker lokaal de bestanden editeren (i.e. vervolledigen, corrigeren, etc.). Na editering verschilt de lokale kopie van het bestand uiteraard van de versie op de centrale server. Het synchroniseren met de server gebeurt via de **commit**-operatie. De aangepaste versie wordt hierbij geupload naar de server. De server houdt de verschillen met de vorige versie bij (**diff**): de vorige versie wordt niet simpelweg overschreven. Via **diff** wordt het verschil met de vorige versie bepaald en dit wordt opgeslagen op de server. Het bestand krijgt dan een nieuw versie-nummer, door incrementering van het vorige versie-nummer.

Bij een **commit**-operatie hoort steeds het door de gebruiker opgeven van een log-boodschap, waarin de gebruiker geacht wordt om de aangebrachte veranderingen kort tekstueel te beschrijven.

Vermits een server alle versies bevat, laat een CVS systeem browsing door alle versies toe, waarbij telkens de verschillen gehighlight worden en de log-boodschappen getoond

worden.

Figuur 17.2 toont de werking van de **commit**-operatie in een CVS systeem.



Figuur 17.2: Principe van **commit**-operatie in een CVS systeem.

17.2.3 update-operatie

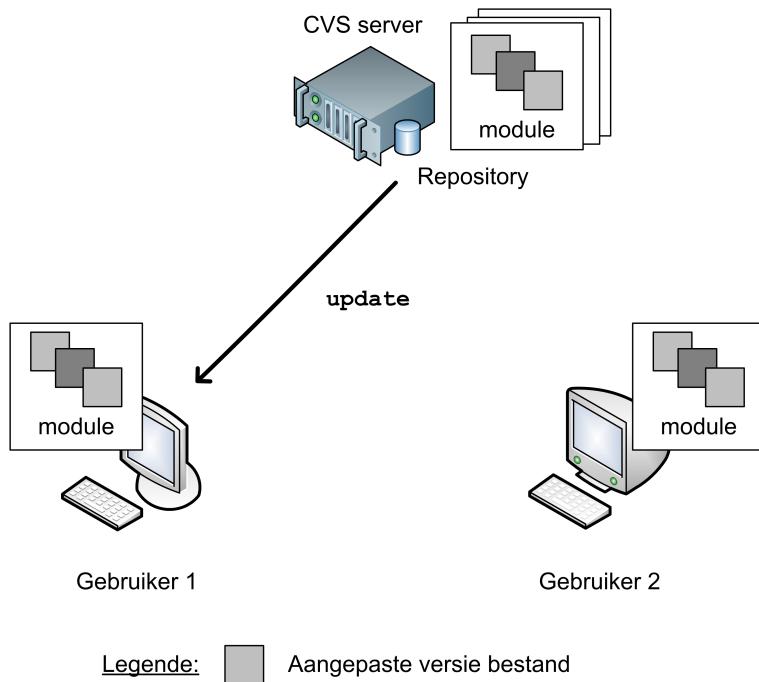
Na een **commit**-operatie, bevat de PC van de gebruiker die de aanpassing heeft aangebracht, de laatste versie van een bestand. Andere actieve gebruikers hebben dan echter nog een vorige versie staan. Via een **update**-operatie, kunnen zij hun bestanden weer synchronizeren om ervoor te zorgen dat ze van elk bestand de laatste versie hebben. Het is dus belangrijk dat een gebruiker eerst een **checkout** of **update** van een module doet, vooral als hij aanpassingen aan een module wil doorvoeren.

De **update**-operatie in een CVS systeem wordt getoond in figuur 17.3.

17.2.4 Oplossen van conflicten

Vermits verschillende ontwikkelaars tegelijkertijd aan hetzelfde bestand kunnen werken, kunnen er mogelijk conflicten optreden. Deze worden gedetecteerd door het CVS systeem. Er zijn twee gevallen:

1. de aanpassingen vonden plaats in verschillende secties van het bestand (bijvoorbeeld verschillende functies in een *.c bestand): er is geen enkel probleem en bij de beide **commit**-operaties worden de aanpassingen allebei doorgevoerd.

Figuur 17.3: Principe van `commit`-operatie in een CVS systeem.

2. de aanpassingen vonden plaats in overlappende secties van het bestand (bijvoorbeeld in dezelfde functie in een `*.c` bestand werd door twee ontwikkelaars aangepast): bij de eerste `commit` wordt de aanpassing doorgevoerd, maar bij de tweede `commit` wordt er een conflict gemeld aan de tweede ontwikkelaar, die dan de keuze moet maken welke aanpassingen effectief dienen doorgevoerd te worden.

Om conflicten te vermijden, wordt meestal mondeling of via e-mail tussen de ontwikkelaars afgesproken in welke volgorde men een aanpassing doorvoert. Na een `commit` van de eerste ontwikkelaar, voert de tweede een `update` uit, waardoor geen conflicten kunnen ontstaan.

17.3 Goed gebruik van CVS

Volgende regels worden best in acht genomen om het gebruik van een CVS systeem vlot te laten verlopen:

- doe een `checkout` van een module, vooraleer bestanden van de module aan te passen,
- doe een `commit` onmiddellijk na een aanpassing,
- voeg steeds een log-boodschap toe, die bondig beschrijft welke aanpassing is doorgevoerd.

Bij het creëeren van een nieuwe module of nieuw bestand, houdt men best rekening met volgende aanbevelingen:

- plan een duidelijke folder-structuur, bijvoorbeeld een aparte folder met bronbestanden (met als naam bijvoorbeeld `src`), aparte sub-folder voor verschillende types van bronbestanden, aparte folder met documentatie (met als naam bijvoorbeeld `man`, Eng.: manual), aparte folder met figuren en schema's (met als naam bijvoorbeeld `fig`),
- kies telkens een duidelijke en veelzeggende naam voor een module en bestand,
- een nieuwe module wordt normaal gezien enkel gecommit, wanneer een eerste stabiele versie beschikbaar is.

Voor de volledigheid geven volgende regels voorbeelden van verkeerd gebruik van een CVS systeem:

- doe nooit een commit-operatie zonder een log-boodschap (zelfs de log-boodschap *typfout verbeterd* of *typo fixed* is zinvol),
- gebruik CVS voor tekst-gebaseerde bestanden, niet voor uitvoerbare bestanden: voeg gecompileerde versies van bronbestanden niet toe in een module, geen pdf-bestanden, wel bijvoorbeeld `*.tex` (latex bronbestanden),
- plaats niet alle bestanden in één folder: achteraf sorteren is moeilijk, want bestanden verplaatsen of verwijderen lukt niet in een CVS systeem (vermits altijd een diff bijgehouden wordt).

17.4 Tagging, Branching, Merging

17.4.1 Tagging

In een CVS module hebben sommige bestanden meer aanpassingen nodig dan andere (omdat ze bijvoorbeeld complexer zijn of fundamentele gegevens bevatten). Op een zeker ogenblik komt men een stabiele versie en is het belangrijk om de bijhorende versie-nummers van alle bestanden bij te houden: op deze manier kan men eenvoudig een vorige stabiele versie downloaden en deze verder gebruiken. Een CVS systeem laat toe om een *tag* te plaatsen op een module, zijnde een naam, die een relevante betekenis heeft (zoals bijv. `rel-1-1` of `version2010`). Deze *tag* kan gebruikt worden bij een latere *checkout*-operatie, wanneer men terug wil keren naar een bepaalde toestand van een module.

17.4.2 Branching

Men kan ook een aparte tak (Eng.: branch) opstarten, waarbij men revisies kan doen in een aparte tak (zonder dat er revisies merkbaar zijn in de hoofdtak). Dit is bijvoorbeeld

nuttig in volgend geval: er is een bug in een applicatie sinds versie 1.4, en men wil deze bug verwijderen om een patch uit te brengen voor gebruikers van versie 1.4. Ondertussen is wel versie 1.7 al klaar, en men wil eerst iets uitproberen op versie 1.4, zonder dat dit impact heeft op de huidige versie 1.7. Het opstarten van een tak op versie 1.4 is in dit geval dan aangewezen.

17.4.3 Merging

Het is nuttig om aparte takken (door *branching* gecreëerd) soms samen te voegen (Eng.: merging). Bijvoorbeeld wanneer men een bug in versie 1.4 in een aparte tak heeft opgelost en men de aanpassingen ook wil doorvoeren in de huidige versie 1.7 van de hoofdtak.

17.5 Bestandsheading - Sleutelwoorden

Gebruik van een CVS systeem laat toe om CVS sleutelwoorden in een bestand te plaatsen, die door de centrale server telkens ingevuld en aangepast worden. Deze CVS sleutelwoorden worden steeds tussen twee \$-tekens geplaatst, zodat ze door het CVS systeem kunnen opgespoord worden.

```
*****
Filename: $Id$
Author: $Author$
Last Changed: $Date$
Log: $Log$
*****
...
(verdere inhoud van het bestand)
...
```

De betekenis van de CVS sleutelwoorden wordt weergegeven in tabel 17.1.

\$Author\$	de naam van de gebruiker die de laatste revisie deed
\$Date\$	de datum en tijdstempel van de laatste revisie
\$Header\$	een standaard header die het volgende bevat: volledige padnaam van het bestand, het revisie nummer, de datum, en de auteur
\$Id\$	zelfde als \$Header\$, behalve bestandsnaam: niet volledig pad
\$Name\$	naam van de tag om dit bestand uit te checken
\$Log\$	de meegegeven log-boodschap bij laatste revisie
\$Revision\$	het revisie nummer toegekend aan deze versie

Tabel 17.1: CVS sleutelwoorden voor gebruik in heading van bestanden

17.6 CVS implementaties

Volgende implementaties van CVS (deel-)systemen worden vaak gebruikt in de praktijk:

- Tortoise CVS: een shell-geïntegreerde CVS client voor Windows,
- Subversion (SVN): een open-source revisie controle systeem,
- DCVS: een gedistribueerd revisie controle systeem, gebaseerd op CVS,
- Bonsai CVS code beheersysteem: een tool met uitgebreide rapporteringsmogelijkheden,
- Commerciële implementaties: bijv. IBM Rational Clear case, etc.

17.7 Referenties

Volgende URL's bevatten interessante informatie in verband met gebruik en configuratie van een CVS systeem:

- <http://tortoisecvs.sourceforge.net/>
- <http://www.nongnu.org/cvs/>
- http://www.linuxdevcenter.com/pub/a/linux/2002/01/03/cvs_intro.html
- <http://www.linuxdevcenter.com/pub/a/linux/2002/01/17/cvsadmin.html>

Hoofdstuk 18

Make-bestanden

18.1 Situering

Grote software projecten kunnen uit duizenden bestanden bestaan en compilatie van al deze bestanden kan tot enkele uren in beslag nemen. Er zijn enkele aandachtspunten bij het compileren van grote projecten:

- bij aanpassing van enkele bestanden in een project, dienen enkel deze bestanden, tezamen met de bestanden die ervan afhankelijk zijn opnieuw gecompileerd te worden. Twee bestanden zijn afhankelijk van elkaar omdat bijv. het ene bestand het andere includeert (in geval van een `*.h` bestand), of omdat in het ene bestand gebruik gemaakt wordt van functies geïmplementeerd in het andere bestand (`*.c` of `*.cpp` bestand). Objectbestanden (`*.o`) die toch ongewijzigd blijven hoeven niet opnieuw gegenereerd te worden.
- voor ontwikkelaars die niet bij de codering betrokken waren is het moeilijk om achteraf te weten welke bestanden gecompileerd en gelinkt dienen te worden: vaak zijn er verschillende versies van een bestand en het uitproberen van alle combinaties van versies kan veel tijd in beslag nemen en is zinloos werk.
- om een programma met debug-informatie te compileren, dienen alle afzonderlijke bestanden met debuginformatie gecompileerd te worden. Het is handig als men verschillende versies van een programma kan vastleggen (met debug informatie, zonder debug informatie, geoptimaliseerde uitvoering, etc.) en bij compilatie en linking dan automatisch de juiste compiler-opties gekozen worden.
- bij code die voor meerdere platformen kan gecompileerd worden, zijn er steeds platformafhankelijke bestanden of code-onderdelen. Het is handig als kan vastgelegd worden welke bestanden of onderdelen van bestanden bij een bepaald platform horen en op welke manier de applicatie voor dit platform kan gecompileerd en gelinkt worden.

Het programma `make` is een belangrijk hulpmiddel bij compilatie en linking. Een make-bestand is het configuratie-bestand, dat door het `make` programma telkens ingelezen

wordt.

Het programma **make** zorgt voor het volgende (elk punt komt hierbij overeen met een aandachtspunt hierboven):

- er wordt bepaald welke bestanden afhankelijk zijn van elkaar en op basis van de tijdsstempel (Eng.: time stamp) van de bestanden wordt bepaald dat als een bestand afhangt van een recenter gewijzigd bestand, dit eerste bestand ook opnieuw dient gecompileerd te worden. In een make-bestand (Eng.: make file) worden de afhankelijkheden tussen de bestanden vastgelegd.
- in het make-bestand wordt vastgelegd welke bestanden dienen gecompileerd te worden voor een applicatie.
- in het make-bestand kunnen compiler opties vastgelegd worden, die geldig zijn voor het compilatie- en linking proces van de applicaties.
- in het make-bestand kunnen verschillende doelen (Eng.: targets) vastgelegd worden. Elk doel komt bijvoorbeeld overeen met een bepaald platform. Ook debug-versies of geoptimaliseerde versies kunnen als doel in het make-bestand opgenomen zijn.

Deze functionaliteiten worden nu verder in dit hoofdstuk in detail behandeld.

18.2 Compiler opties

Om een compilatieproces te kunnen aansturen, dient men de commandolijn opdrachten voor compilatie en linking te specifiëren. Er zijn verschillende C en C++ compilers beschikbaar, bijvoorbeeld **cc**, **CC**, **gcc** en **g++**. Deze worden voornamelijk in Unix/Linux omgevingen gebruikt.

Voorbeelden van dergelijke opdrachten zijn:

```
CC -o application main.c adt.c
gcc -o application adt.c main.c
g++ data.cpp main.cpp
```

De eerste regel geeft de opdracht om twee bestanden **adt.c** en **main.c** te compileren tot respectievelijk **adt.o** en **main.o**, welke dan via de linker naar een uitvoerbaar bestand omgezet worden. De [-o] optie specificeert de naam van het uitvoerbaar bestand. In de tweede regel worden dezelfde bestanden gecompileerd en gelinkt, maar wordt een andere compiler gebruikt. Als de [-o] optie niet aanwezig is (zoals in de derde regel), wordt de default naam **a.out** gebruikt.

Indien enkel compilatie en geen linking vereist is, wordt de optie [-c] gebruikt:

```
g++ -c -o BinaryTree.o BinaryTree.cpp
```

Deze opdracht genereert het objectbestand **BinaryTree.o**. Als de [-o] optie niet aanwezig is, wordt de naam geconstrueerd door de extensie van het bronbestand te veranderen

naar *.o. De optie `-o BinaryTree.o` is dus in dit voorbeeld niet strikt nodig. Enkel linken (wanneer de objectbestanden reeds beschikbaar zijn) kan op de volgende manier:

```
gcc -o application adt.o main.o
```

Bemerkt dat bovenstaande opties gelden zowel voor compilatie en linking van zowel C en C++ bestanden (en ook van Objective-C bestanden, zoals in hoofdstuk 22 aan bod komt).

Andere zinvolle opties worden weergegeven in tabel 18.1. Via het `man` commando (afkorting van *manual*) kan men een overzicht krijgen van alle beschikbare opties van een compiler, bijvoorbeeld:

```
man gcc
```

<code>-Wall</code>	toon alle waarschuwingen (warnings)
<code>-g</code>	toevoegen van debugger informatie
<code>-I dir</code>	zoekt eerst in directory <code>dir</code> naar te includeren bestanden
<code>-lxxx</code>	linkt een library (naam: <code>libxxx.a</code>)
<code>-L dir</code>	zoekt library files in directory <code>dir</code>

Tabel 18.1: Compiler en linker opties van een command line compiler.

18.3 Programma make en make-bestanden

Het programma `make` bepaalt automatisch welke stukken code (opnieuw) moeten 'gemaakt' worden. Het voert hiervoor de nodige compilatie en linking opdrachten uit en zorgt ervoor dat automatisch enkel bestanden opnieuw gecompileerd worden op basis van andere bestanden indien deze gewijzigd zijn.

Een make-bestand bevat de opdrachten en wordt door de gebruiker opgegeven. Het beschrijft de relaties tussen de bestanden en geeft regels voor het updaten van de bestanden.

Hierbij wordt een voorbeeld gegeven van de inhoud van een make-bestand:

```
application: adt.o main.o
    gcc -o application adt.o main.o
adt.o: adt.c adt.h
    gcc -c -o adt.o adt.c
main.o: main.c test.h
    gcc -c -o main.o main.c
```

na de dubbele punt (:) worden steeds de afhankelijke bestanden vermeld (gescheiden door spaties) om het bestand links van de dubbele punt (:) te kunnen maken. De regel eronder begint **steeds** met een tabulatie-karakter (belangrijk: anders wordt er een foutbericht gegenereerd en wordt het bestand in kwestie niet gemaakt) en vervolgens

wordt de volledige opdracht gegeven om het linkerbestand te genereren op basis van de rechterbestanden.

De inhoud van een make-bestand wordt opgeslagen in het bestand met naam: **Makefile** (geen extensie). Het compilatie- en linking proces wordt dan gestart door de opdracht **make** op de commandolijn:

```
$ make
```

Indien men een andere naam kiest, dient men deze via de **-f** optie (Eng.: file) aan de make opdracht toe te voegen (zodat het juiste bestand ingelezen en uitgevoerd wordt).

18.4 Variabelen

Men kan ook variabelen toevoegen aan een make-bestand. Typisch gebeurt dit bovenaan het bestand voor keuze van de compiler en compilatie- en linking opties. Onderstaand voorbeeld van een make-bestand illustreert dit:

```
CC := gcc
CFLAGS := -g Wall
LFLAGS :=
all: application
adt.o: adt.c adt.h
        $(CC) -c adt.c $(CFLAGS)
main.o: main.c test.h
        $(CC) -c main.c $(CFLAGS)
application: adt.o main.o
        $(CC) -o application adt.o main.o $(LFLAGS)
```

de variabele **CC** stelt de naam van de compiler voor en kan in het make-bestand gebruikt worden door **\$(CC)** te gebruiken (of in het algemeen **\$(<variabele_naam>)**). De variabele **CFLAGS** stelt de compiler opties voor (ook wel compiler vlaggen genoemd) en analoog stelt **LFLAGS** de linker opties voor.

Bemerk dat in geval van lange regels, men deze ook kan opsplitsen over twee regels, door gebruik van een \ teken, zoals hieronder geïllustreerd wordt voor de laatste regel van bovenstaand make-bestand:

```
application: adt.o main.o
        $(CC) -o application adt.o main.o \
        $(LFLAGS)
```

18.5 Automatische variabelen

Automatische variabelen kunnen in het make-bestand gebruikt worden om te vermijden dat men telkens bestandsnamen van de eerste regel op de tweede regel moet herhalen. De mogelijke automatische variabelen worden opgesomd in tabel 18.2.

\$@	doel (van de regel)
\$<	eerste bronbestand
\$	alle bronbestanden met spaties ertussen
\$?	alle bronbestanden die recenter zijn dan het doel, met spaties ertussen

Tabel 18.2: Automatische variabelen in make-bestanden.

Hieronder wordt een voorbeeld getoond van het gebruik van automatische variabelen in een make-bestand:

```
# Dit is een regel commentaar
CC := gcc
CFLAGS := -g -Wall # c-compiler opties
LDFLAGS := # linker opties
all: application
adt.o: adt.c adt.h
    $(CC) -c $< $(CFLAGS)
main.o: main.c test.h
    $(CC) -c $< $(CFLAGS)
application: adt.o main.o
    $(CC) -o $@ $^ $(LDFLAGS)
```

Bemerkt dat commentaar aan de hand van het # gespecificeerd wordt.

De waarde van variabelen kan ook via de commandolijn doorgegeven worden: deze hebben dan voorrang op variabelen, die in het bestand zelf gespecificeerd werden.

18.6 Condities

Voorwaardelijke constructies worden vaak gebruikt in make-bestanden. Hieronder wordt een voorbeeld gegeven:

```
COMPILER := SUN_CPP
#if equal syntax: ifeq (arg1, arg2)
ifeq ($(COMPILER), SUN_CPP)
    CC := CC
    # meerdere instructies mogelijk
else
    CC := gcc
endif
application: main.c
    $(CC) -o application main.c $(FLAGS)
```

Waarden van variabelen kunnen doorgegeven worden via de commandolijn:

```
make FLAGS=-g
```

Deze kunnen door het make-bestand niet veranderd worden, bijv. de opdracht *compileer met GNU compiler in debugmode* komt overeen met:

```
make COMPILER=SUN_GNU FLAGS=-g
```

Testen op (on-)gelijkheid kan aan de hand van volgende constructies:

```
ifeq (arg1, arg2)
ifneq (arg1, arg2)
```

Testen op (on-)gedefineerd zijn van variabelen kan aan de hand van volgende constructie:

```
ifdef var
ifndef var
```

Een variabele is gedefinieerd wanneer ze een niet-lege waarde heeft

```
CPP := # NIET gedefineerd
CPP := 0 # gedefineerd
```

18.7 Doelen - Eng.:Targets

Een make-bestand maakt het mogelijk om verschillende doelen (Eng.: targets) vast te leggen. Bij het starten van het programma `make` wordt dan het doel gespecificeerd. Twee voorbeelden van doelen zijn:

- verschillende versies van een applicatie, bijv. een debugversie, een geoptimaliseerde versie voor een specifiek toestel, etc. Elke versie komt dan typisch overeen met een doel in het make-bestand.
- het verwijderen van alle objectbestanden en uitvoerbare bestanden, zodat alles opnieuw dient gemaakt te worden.

Hieronder wordt een make-bestand getoond, met twee doelen, namelijk `all` en `clean`.

```
CC := CC
RM := rm
all: application
clean:
        $(RM) application *.o
application: main.c
        $(CC) -o application main.c $(FLAGS)
```

Door het uitvoeren van de volgende opdracht op de commandolijn:

```
$ make clean
```

worden alle gemaakte bestanden verwijderd en via:

```
$ make all
```

wordt de volledige applicatie opnieuw gemaakt.

18.8 Standaardregels

Om te vermijden dat telkens de afzonderlijke bestandsnamen dienen vermeld te worden in de make-bestanden, kan men ook standaardregels specifiëren, die geldig zijn voor een groot aantal van de bestanden. Gebruik van de standaardregels resulteert dus in kortere make-bestanden (en dus minder kans op fouten of vergeten regels).

De standaardregels worden opgebouwd aan de hand van het %-teken, dat links en rechts van de dubbele punt (:) voorkomt.

Ter illustratie wordt hieronder een voorbeeld make-bestand gegeven:

```
% .o : %.c
    $(CC) -c $< $(CFLAGS)
% .o : %.cpp
    $(CC) -c $< $(CFLAGS) $(CPPFLAGS)
```

Het betekent dat een `.o` bestand steeds geconstrueerd wordt door het corresponderende `.c` of `.cpp` bestand met dezelfde naam te compileren. Belangrijk hier is dus dat object-bestanden dezelfde naam krijgen als hun corresponderende bronbestanden.

18.9 Generische make-bestanden

Door combinatie van de bovenstaande regels, kan men automatisch make-bestanden genereren, die geldig zijn voor alle projecten. Hieronder wordt hiervan een uitgebreid voorbeeld gegeven.

Enkel de bovenste regels van het bestand dienen aangepast te worden per project en zijn dus project-specifiek.

```
# Generische makefile
#
#COMPILER := SUN_GNU
#COMPILER := SUN_CPP
COMPILER := DOS_GNU
PROG := test # do not include a file extension
OBJECTS := test.o test1.o test2.o # module names
CPP := # set to 1 to use C++
RTTI := # set to 1 to use RTTI
MATH := # set to 1 to use mathematics lib

# Automated part
# DO NOT MODIFY BELOW THIS LINE
# -----

# Compiler options
MYCFLAGS :=

# Linker options
MYLDFLAGS :=
```

```

ifeq ($(COMPILER), SUN_GNU)
ifdef CPP
    CC := g++
else
    CC := gcc
endif
RM := rm
ifdef MATH
    # equivalent: # MYCFLAGS := $(MYCFLAGS) -lm
    MYLDFLAGS += -lm
endif
endif

ifeq ($(COMPILER), SUN_CPP)
    CC := CC
    RM := rm
ifdef RTTI
    MYCFLAGS += -features=rtti
endif
ifdef MATH
    MYLDFLAGS += -lm
endif
endif

ifeq ($(COMPILER), DOS_GNU)
ifdef CPP
    CC := g++
else
    CC := gcc
    RM := del
    PROG := $(addsuffix .exe,$(PROG))
endif

all: $(PROG)

%.o : %.c
    $(CC) -c $< $(MYCFLAGS) $(CFLAGS)

%.o : %.cpp
    $(CC) -c $< $(MYCFLAGS) $(CXXFLAGS)

$(PROG): $(OBJECTS)
    $(CC) -o $@ $^ $(MYLDFLAGS) $(LDFLAGS)

# Remove all output files
# Needed if only a headerfile is changed

clean:

```

Hoofdstuk 18: Make-bestanden

```
ifeq ($(COMPILER), DOS_GNU)
    $(RM) *.o
    $(RM) $(PROG)
else
    $(RM) *.o $(PROG)
endif
```


Hoofdstuk 19

Hedendaags Belangrijke Software Technologieën

In dit hoofdstuk worden belangrijke software technologieën kort toegelicht. Deze worden in verdere cursussen in meer detail uitgewerkt.

19.1 Java Technologieën

19.1.1 Standaard Editie versus Enterprise Editie

Naast de Java Standaard Editie (Eng.: Java Standard Edition), afgekort Java SE, is ook Java Enterprise Editie (Eng.: Java Enterprise Edition), afgekort Java EE, een belangrijke technologie. Java SE laat toe om applets, standalone applicaties en gedistribueerde applicaties (i.e. waarbij de objecten op verschillende machines elkaars methoden kunnen oproepen) te ontwerpen. Java EE wordt gebruikt voor ontwerp van applicaties die een grote onderliggende databank gebruiken en via een Web-applicatie met de gebruikers interageren. Een belangrijk voorbeeld van een dergelijke applicatie is Amazon.com, een zogenaamde eCommerce (Eng.: electronic Commerce) applicatie.

Java EE applicaties maken gebruik van een applicatieserver, die o.a. volgende functionaliteit aanbiedt: automatisch activeren van de componenten wanneer deze opgeroepen worden, automatisch stoppen van componenten wanneer deze een bepaalde tijd niet opgeroepen werden, automatische synchronisatie van de gegevens met de databank (zodat er geen SQL opdrachten dienen geschreven te worden) en transactiebeheer.

Vele industriële applicaties maken gebruik van een applicatieserver, waardoor de ontwikkeltijd van deze applicaties beperkt wordt. De applicatieserver zorgt echter voor bijkomende uitvoeringstijd en kan, indien foutief geconfigureerd, een traag werkende of falende applicatie veroorzaken.

19.1.2 Prestatie evaluatie

Voor de evaluatie van de prestatie van een applicatie zijn twee metrieken belangrijk:

1. antwoordtijd (Eng.: response time), i.e. de tijd om een aanvraag (Eng.: request) af te handelen,
2. doorvoersnelheid (Eng.: throughput), i.e. het aantal aanvragen dat per seconde door de applicatie kan verwerkt worden.

Een aanvraag wordt ook dikwijls een oproep (Eng.: call) genoemd en een veelgebruikt synoniem voor antwoordtijd in deze context is vertragingstijd (Eng.: latency).

De prestatie van een applicatie wordt meestal in een grafiek uitgezet, met op de X-as het aantal oproepen per seconde (Eng.: calls per second, afgekort caps) en op de Y-as de volgende meetpunten corresponderend met de x-waarden: de gemiddelde antwoordtijd, de 50-percentiel van de antwoordtijd, de 95-percentiel van de antwoordtijd, en de CPU belasting (Eng.: load). In sommige kritische applicaties wordt de 99-percentiel ook uitgezet. Door een bovenlimiet voor de Y-waarden te specifiëren, kan eenvoudig de maximale caps van de applicatie bepaald worden.

Bij sommige applicaties is een lage antwoordtijd cruciaal voor de gebruikers (deze applicaties worden *low latency* applicaties genoemd), bij andere applicaties is het essentieel dat zeer veel gelijktijdige aanvragen kunnen verwerkt worden (deze applicaties worden *high throughput* applicaties genoemd).

19.1.3 JVM tuning

De prestatie van een Java applicatie (zowel Java SE als Java EE) hangt af van de instellingen van de Java Virtuele Machine (JVM), namelijk:

1. de JVM geheugenstructuur: deze kan ingesteld worden afhankelijk van de noden van de applicatie,
2. de garbage collector opties: deze laten toe om de garbage collection in parallel te laten uitvoeren (Eng.: multi-threaded) en tegelijkertijd (Eng.: concurrent) met de uitvoering van de applicatie.

Door deze beide JVM-instellingen op de juiste manier te kiezen (Eng.: tuning) kan de prestatie van een Java applicatie aanzienlijk verhoogd worden.

19.2 Middleware

Middleware wordt gebruikt om gedistribueerde applicaties (i.e. applicaties die verspreid zijn over verschillende computers of toestellen) te ontwikkelen. De naam middleware komt van het feit dat het zich situeert tussen de hardware en de software: het laat de software-ontwikkelaars toe om abstractie te maken van (i) de onderliggende hardware en (ii) het feit dat de objecten in een object-georiënteerde applicatie zich op verschillende locaties kunnen bevinden.

Middleware wordt vaak gebruikt voor aansturing van robotten, sensoren en actuatoren. In bijvoorbeeld ook de luchtvaartindustrie, spoorwegindustrie of procesautomatisatie wordt veelal beroep gedaan op middleware.

19.3 Web services

Web services is een belangrijke technologie bij de ontwikkeling van gedistribueerde applicaties. Centraal bij web services is het gebruik van XML, hetgeen gebruikt wordt om:

1. de interface te beschrijven van de web service, i.e. de aangeboden methoden, hun argumenten, de return waarde, en het onderliggende protocol (bijv. HTTP),
2. de parameters bij een invocatie door te geven en het resultaat naar de oproeper terug te geven,
3. de protocol boodschappen te sturen ter ontdekking (Eng.: discovery) van de beschikbare web services.

Web services worden zeer vaak in B2B (Eng.: Business-to-Business) context gebruikt, i.e. koppeling van applicaties tussen bedrijven. Een voorbeeld is de koppeling tussen een reservatie-applicatie in een reisbureau en deze in luchtvaart-maatschappijen en hotelketens.

19.4 Ruby

Ruby is een geïnterpreteerde programmeertaal, geïnspireerd op Java en de Perl scripting taal. Het werd in de jaren '90 bedacht en heeft sinds 2005 opgang gemaakt, vooral dankzij Ruby-on-Rails hetgeen toelaat om snel Web-gebaseerde Ruby applicaties te ontwikkelen. Enkele eigenschappen van Ruby zijn:

- het is een object-georiënteerde taal met ondersteuning voor overerving,
- reguliere expressies voor bijv. string-matching of opsplitsen van strings in onderdelen (Eng.: string tokenizing) kunnen efficiënt opgegeven worden (net zoals in Perl),
- een blok code kan als argument aan een methode doorgegeven worden, deze blok code wordt dan telkens opgeroepen waar aangeduid in de methode;
- type controle is flexibeler dan in C++ en Java: een object kan als argument aan een methode doorgegeven worden, ook al verwacht de methode een object van een andere klasse, op voorwaarde dat de beide klassen een corresponderende methode hebben.

19.5 Aspect-oriëntatie

Aspect-oriëntatie is ontstaan uit de volgende vaststelling: in zeer veel applicaties zijn er herhalingen van code op specifieke plaatsen, bijv. voor logging van gegevens, beveiliging (bijv. authenticatie-code, autorisatie-code), etc. Door herhaling van code is het niet

evident om de code op de verschillende plaatsen perfect gelijk te houden, zeker in geval er updates vereist zijn.

De aanpak die bij aspect-oriëntatie gevuld wordt is de volgende:

1. in de code worden punten aangegeven waar de aanpassingen dienen te komen (Eng.: *join points*),
2. een apart programma zorgt voor invoegen van de code, dit invoegen wordt *weaving* (Eng.: weaving) genoemd.

Wat betreft de weaving zijn er twee opties: static weaving (door de compiler) en dynamic weaving (at run-time). AspectC++ en AspectJ zijn implementaties in respectievelijk C++ en Java, welke frequent gebruikt worden.

19.6 Software voor mobiele toestellen

Ontwikkeling van software voor mobiele toestellen (PDA's, smart phones, tablets) wordt steeds belangrijker. Een belangrijke vaststelling is dat er zeer veel verschillende besturingssystemen in omloop zijn en een eigen software-ontwikkelplatform voor elk type toestel hebben.

We onderscheiden volgende belangrijke categorieën van software-ontwikkelplatformen voor mobiele toestellen:

1. Java ME (Eng.: Micro Edition): vereist een JVM op het mobiele toestel en laat toe om applicaties te ontwikkelen die platform-onafhankelijk en dus porteerbaar zijn (naar toestellen die ook een JVM kunnen draaien).
2. Android: een Linux-gebaseerd platform gelanceerd door Google op vooral HTC toestellen. Programmering van applicaties gebeurt in Java met de Android specifieke Java SDK, of in C of C++ met de Android Native Development Kit (NDK).
3. .Net: is het platform voor toestellen met een Windows Mobile besturingssysteem. Ontwikkeling van code gebeurt in C# en met bijvoorbeeld Visual Studio als IDE.
4. Symbian C++: wordt voornamelijk gebruikt op Nokia toestellen. Symbian is het besturingssysteem van Nokia. Symbian C++ is gebaseerd op C++, maar is specifiek voor toestellen met beperkt geheugen en processing mogelijkheden. Er wordt bijvoorbeeld steeds expliciet met een stack gewerkt om gegevens bij te houden of door te geven, waardoor de code op het eerste zicht cryptischer oogt dan C++ code.
5. iPhone, iPod, iPad: ontwikkeling van code gebeurt in Objective-C, gebaseerd op de programmeertaal C. Hoofdstuk 22 is volledig gewijd aan de programmeertaal Objective-C.

Hoofdstuk 20

Ontwikkeling van Betrouwbare Software

In dit hoofdstuk worden belangrijke hedendaagse principes toegelicht, die ervoor zorgen dat de ontwikkelde software voldoet aan de eisen van de klant, grondig getest en opgevolgd kan worden, en bovendien efficiënt binnen een tijdsbestek kan opgeleverd worden. Deze zorgen er dus voor dat betrouwbare software applicaties kunnen gebouwd, afgeleverd en onderhouden worden.

20.1 BDD (Eng.:Behavior Driven Design)

Dit is een belangrijk onderdeel van het agile programming ontwikkelingsproces. Het benadrukt het belang van de verhalen van de gebruikers (Eng.:user stories) hoe ze exact hun applicatie willen gebruiken en welk gedrag (Eng.:behavior) de applicatie dient te vertonen bij gebeurtenissen uit de *user stories*. Deze *user stories* worden dan gebruikt om prototypes van gebruikerinterfaces te bouwen en aan de hand van verhaalborden (Eng.: storyboards) worden de vereisten van de gebruikersinterfaces vastgelegd. Aan de hand van tools kunnen de *user stories* omgezet worden naar acceptatie testen (Eng.:acceptance tests). BDD wordt dikwijls in combinatie met TDD (Eng.:Test Driven Development) toegepast.

20.2 SPLE (Eng.:Software Product Line Engineering)

Als deze aanpak gevuld wordt, wordt de software gemodelleerd als een collectie van *features*. Door features te selecteren of te deselecteren kunnen verschillende software varianten worden gecreëerd. Van de features wordt een model opgesteld, die ook de afhankelijkheden tussen de features bijhoudt. Dit principe is geïnspireerd op de aanpak in de auto-industrie: de klanten kunnen de gewenste opties kiezen en het aantal mogelijke opties is op voorhand bepaald. Op die manier hebben de klanten een overzicht van alle mogelijkheden. Dit komt de oplevertijd ten goede en het vermindert dat de klanten onrealistische eisen stellen.

20.3 Software as a Service - SaaS

Dankzij cloud computing worden heel veel software-applicaties nu in een datacenter uitgevoerd in plaats van lokaal bij de klanten. Eenzelfde software-applicatie kan dan dienen om meerdere klanten tegelijkertijd te bedienen. De klanten kunnen inloggen op hun applicatie en concrete configuraties voor hun applicatie aanbrengen. Dit principe is momenteel zeer populair en heel veel software ontwikkelingsprocessen concentreren zich op het *Software as a Service* model.

Indien men enkel interesse heeft om eigen software op een cloud infrastructuur uit te voeren is het IaaS (Eng.:Infrastructure as a Service) model aangewezen (bijv. Amazon of Rackspace). Indien men software voor cloudomgevingen wil ontwikkelen en wil gebruik maken van een bestaand platform en ontwikkelomgeving, is het PaaS (Eng.:Platform as a Service) model aangewezen (bijv. Google App Engine).

20.4 Ontwikkelingsproces

20.4.1 Scrum

Scrum is een populair voorbeeld van een agile software ontwikkelingsproces. Het legt de nadruk op teamwerk en nauwe samenwerking tussen de ontwikkelaars (zelfde fysische locatie en regelmatige interacties). Een belangrijk principe is dat gedurende het project de klanten hun vereisten kunnen bijsturen en dat er snel kan op ingespeeld worden.

Er worden regelmatig *sprints* gedefinieerd: *sprints* zijn een tijdsgelimiteerd inspanning om een concrete doelstelling te bereiken, bijvoorbeeld een eerste iteratie van een prototype, of een geïntegreerde applicatie. Er zijn verschillende types van meetings: de sprint planning meeting, de dagelijkse scrum meeting, en twee eindmeetings (sprint review meeting en sprint retrospective meeting: de review meeting bespreekt het resultaat van de sprint met de ontwikkelaars en de klanten, in de retrospective meeting wordt het verloop van de sprint besproken en wat er kan aangepast worden om de volgende sprint beter te laten verlopen).

20.4.2 Tools voor code validatie

Het is handig als de gebruikte tools voor softwareontwikkeling onmiddellijk feedback geven als er gevraaglijke of niet correcte constructies gebruikt worden. Een belangrijk voorbeeld is de combinatie van javascript en google closure.

20.4.3 Geautomatiseerd testen

Het is belangrijk dat geschreven code grondig kan getest worden en dat dit testen niet manueel dient te gebeuren, maar geautomatiseerd verloopt. Belangrijke principes in deze context zijn Unit Testing en TDD (Eng.:Test Driven Development).

20.4.4 Issue and project tracking software

In hedendaagse software ontwikkelingsprocessen maakt men steeds gebruik van issue and project tracking software. Een belangrijk voorbeeld hiervan is JIRA.

20.4.5 Documentatie-beheer

Een goede strategie voor documentatie-beheer is eveneens belangrijk. Echter, het wordt aangeraden om teveel nadruk leggen op het rigoureus en tijdrovend documenteren van projecten. Dikwijls wordt voor een Wiki-gebaseerde oplossing gekozen, *Redmine* is momenteel ook een populaire tool.

20.4.6 Continue Integratie

CI (Eng.:Continuous Integration) is een principe van extreme programming, waarbij code van ontwikkelaars meerdere malen per dag geïntegreerd wordt. Op deze manier wordt snel ingespeeld op integratieproblemen. Het wordt frequent gebruikt in combinatie met geautomatiseerde testen (i.e. de code wordt enkel geïntegreerd als de testen succesvol verlopen zijn). Een veelgebruikte tool voor continue integratie is *Jenkins*.

20.4.7 Release workflows

Het is belangrijk om op voorhand te bepalen hoe de aflevering van software zal verlopen, i.e. welke stappen doorlopen worden vooraleer de applicatie vrijgegeven (Eng.:released) wordt aan de klanten. Dit wordt aangeduid door de term release workflow, en bevat uitgebreide testen, controles en reservatie van de tijd van een team van ontwikkelaars en testers.

20.5 Eigenschappen van goed software ontwerp

20.5.1 Data aggregation

Als de applicatie veel data verzamelt, is het belangrijk dat de data zodanig opgeslagen wordt zodat ze nadien snel kan gevraagd worden. Als men bijvoorbeeld elke seconde data verzamelt, dan is men na een tijdje enkel geïnteresseerd in het gemiddelde/minuut, en is er verder in de tijd slechts interesse in het gemiddelde/uur, gemiddelde/dag, gemiddelde/week, etc. Het is dus nuttig om dit vooraf al te voorzien, zodat men bij een eenvoudige bevraging geen uitgebreide berekeningen meer hoeft uit te voeren (bijvoorbeeld het gemiddelde over een maand berekenen).

20.5.2 Zero configuration

De software-applicatie moet direct werken zonder extra configuratiewerk. Hiervoor is het belangrijk dat er goede default waarden gebruikt worden voor de configuratie-parameters en dat de applicatie ook zelf de omgeving kan ontdekken (Eng.: auto-discovery).

20.5.3 Daemon monitoring and startup

Meestal bestaan software-applicaties uit verschillende processen, die elk simultaan actief dienen te zijn. Deze processen worden ook dikwijls *daemons* genoemd. Het is belangrijk dat er automatisch gecontroleerd wordt of alle *daemons* nog actief zijn, en indien er een *daemon* niet meer actief zou zijn, dat deze automatisch opnieuw opgestart wordt. Dit verhindert een langdurige faling van een applicatie omdat bijvoorbeeld één of meerdere *daemons* niet meer actief zouden zijn en dit niet snel opgemerkt wordt.

20.5.4 Dashboard applicatie

Het is belangrijk dat er met één oogopslag de toestand van een applicatie kan bekijken worden. Een *dashboard* applicatie visualiseert de belangrijke metrieken van een applicatie, bijv. het aantal succesvolle operaties, het aantal falingen, de antwoordtijd van aanvragen. Bovendien toont het ook de statistische verwerking van de metrieken (bijv. gemiddelden, standaard afwijkingen, *worst case* waarden, *best case* waarden, etc.)

20.5.5 Product feedback

Het is zeer handig als een applicatie bij een klant zelf statistieken kan verzamelen over de prestatie van de applicatie, de populariteit van de applicatie en de verschillende onderdelen. Op deze manier kan men goed inschatten welke uitbreidingen of aanpassingen voor de applicatie nuttig kunnen zijn. Het automatisch en regelmatig doorsturen van deze feedback vereist de goedkeuring van de klant.

Hoofdstuk 21

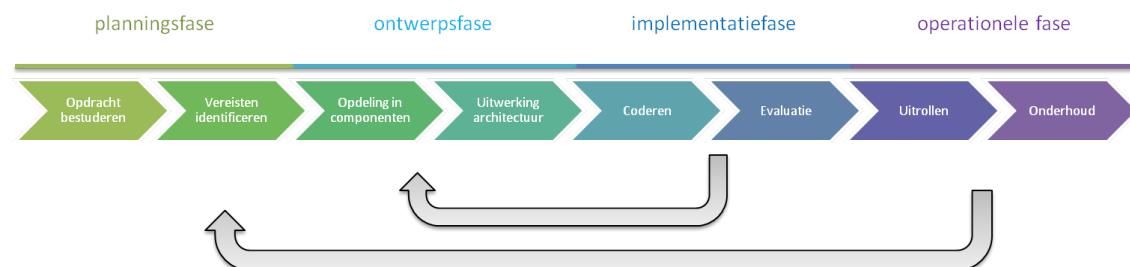
Software ontwikkeling in de praktijk

Software ontwikkeling is meer dan een programmeertaal beheersen. Er zijn duidelijke verschillen tussen bijvoorbeeld JAVA, C en C++. Maar er zijn zeker ook fundamentele gelijkenissen aan het ontwikkelen van software in om het even welke programmeertaal. De doelstelling van dit hoofdstuk is om een overzicht te geven van de verschillende stappen die je als software ontwikkelaar doorloopt, beginnend bij het krijgen van de opdracht omschrijving tot het afleveren van het software-product: hoe eraan te beginnen, waarop moet men letten, wat zijn typische concepten, wat zijn best practices voor bepaalde problemen, welke stappen moeten gevuld worden.

Verder komt ook collaboratie in software ontwikkeling aan bod: samenwerken aan een project is eerder regel dan uitzondering in grote software projecten. Samenwerken aan code, is niet hetzelfde als samenwerken aan een tekstdocument. Verder in dit hoofdstuk wordt dan ook even stil gestaan bij collaboratie, versioning, tooling en tips.

21.1 Overzicht Software-ontwikkelingsproces

Het ontwikkelen van software kan opgedeeld worden in een aantal conceptuele stappen. Deze kunnen als goede *guideline* dienen om een opdracht tot een goed einde te brengen. Ontwikkelen van software begint bij het krijgen van die opdracht. Veelal start men direct bij het programmeren zelf, maar dit is zelden een goed idee. Figuur 21.1 toont een



Figuur 21.1: Overzicht software ontwikkelingsproces

schematisch overzicht van de verschillende stappen, die klassiek worden ondergebracht in vier fases (de zogenaamde levenscyclus van de software):

1. **De planningsfase:** het vertalen van een opdracht of idee naar een probleem dat met software kan opgelost worden.
2. **De ontwerpsfase:** het op methodologische manier uitwerken van een abstracte oplossing.
3. **De implementatiefase:** het coderen van het ontwerp en evalueren van de resulterende software aan de hand van verschillende testen.
4. **De operationele fase:** het uitrollen en onderhouden van de software, met daarnaast eventueel het introduceren van nieuwe features.

Let ook op de feedback-lussen vanuit de implementatie en operationele fases. In de praktijk wordt dergelijk stappen-plan bijna nooit integraal van begin tot einde uitgevoerd en zal er een wisselwerking tussen de verschillende fases ontstaan, bv. nieuwe inzichten tijdens het coderen kunnen een effect hebben op het ontwerp of het toevoegen van een feature na uitrolling kan een impact op de initiële lijst van vereisten hebben.

In de volgende secties wordt meer uitleg gegeven voor elk van deze stappen.

21.1.1 Opdracht bestuderen

Heel veel projecten gaan verkeerd, omdat de opdracht niet op voorhand aandachtig gelezen en vooral begrepen werd! Het doel van een opdracht, wat het resultaat moet zijn, kan soms afwijken van wat men op het eerste gezicht verwacht. Ook is het zo dat ‘de klant’ dikwijls zelf niet goed weet wat hij/zij wilt. De omschrijving die gegeven wordt van de oplossing, is niet noodzakelijk de beste oplossing voor het probleem. Onderzoek daarom steeds aandachtig de opgave. Aandachtig lezen van een opdracht en alle bijgeleverde informatie kan veel mogelijke problemen en onduidelijkheden op voorhand verhelpen!

21.1.2 Vereisten identificeren

De opdracht op zich is zelden een directe oplijsting van wat er moet gebeuren. Door aandachtig bestuderen van de opdracht (zoals eerder vermeld), kan (dikwijls tussen de lijnen) ontdekt worden wat de effectieve *vereisten* zijn. Deze vereisten zijn de echte noden van de oplossing die ontwikkeld zal worden, wat moet die kunnen, tot wat moet het in staat zijn, op welke aspecten moet gefocust worden. En misschien ook belangrijk: wat moet het niet kunnen! Het analyseren van de vereisten gebeurt typisch in twee stappen.

Allereerst moeten de echte functionele vereisten opgeliist worden. Dit is dus een lijst met functies die de software moet kunnen. Een soort van gedetailleerde *feature* lijst. Eens deze opgeliist zijn, zal daaruit redelijk snel duidelijk worden aan welke aspecten je

als ontwikkelaar extra aandacht zal moeten besteden: schaalbaarheid, uitbreidbaarheid, aanpasbaarheid, uitvoeringssnelheid, etc. (de zogenaamde *kwaliteitsattributen*)

Typisch zullen al deze aspecten belangrijk lijken en dat is ook zo. Maar er moet een keuze gemaakt worden, want sommige aspecten kunnen elkaar beïnvloeden. Zo zal het gebruik van een framework om modulair te kunnen werken, de uitbreidbaarheid en aanpasbaarheid van de software wel ten goede komen, maar kan de overhead die door dergelijk framework geïntroduceerd wordt een negatieve impact hebben op de prestaties. Dit betekent dat bijvoorbeeld voor een kritische real-time applicatie deze aspecten niet combineerbaar zullen zijn.

De eenvoudigste manier om dit te doen is door een soort van prioriteitenlijst te maken, zodat duidelijk wordt waar de meeste focus op gelegd zal worden. Probeer hier zeker terug te koppelen naar de initiële opdrachtdocumentatie. Welk probleem probeert men op te lossen, dit geeft veelal ook een duidelijk beeld welke aspecten belangrijker zijn dan andere.

Deze functionele en technische vereisten lijsten dienen als basis voor de verdere software ontwikkeling en dus de volgende stap in het ontwikkelingsproces.

21.1.3 Opdeling in componenten

Eens de vereisten duidelijk zijn, is het tijd om *high-level* componenten te identificeren. Dit betekent niet dat er al klassen of interfaces aangemaakt worden. Het gaat hier eerder over grote subsystemen of aspecten die te onderscheiden zijn. Deze high-level componenten aflijnen helpt bij het isoleren van bepaalde aspecten van de software. Dit is over het algemeen een goede manier van werken, omdat alle code logisch gegroepeerd wordt. Het verhoogt ook enorm de modulariteit van de code, waarbij een bepaald subsystem bijvoorbeeld zou kunnen vervangen worden door een alternatieve implementatie zonder de rest van het systeem te breken.

Eens een aantal van die componenten gekend zijn, kan er gekeken worden naar de interactie tussen deze verschillende componenten. *Welke informatie moet van welk naar welk subsystem gestuurd worden? Welke vorm neemt de gestuurde informatie aan?* Een aantal van deze high-level componenten zal ook verder kunnen opgedeeld worden in kleinere subsystemen, wederom met de nodige onderlinge interacties.

De onderverdeling in componenten kan getoetst worden aan de lijst van functionele vereisten. Er wordt best nagegaan voor elke functie of die binnen de afgelijnde subsystemen past, of de communicatie tussen die componenten logisch is. Indien nodig wordt er iteratief geoptimaliseerd.

Vanaf een bepaald niveau van detail, zal deze stap naadloos overgaan in de verdere uitwerking van de architectuur waarbij het ontwerp dan wel tot op het niveau van de

individuele klassen doorgevoerd wordt (zie volgende sectie).

21.1.4 Uitwerking architectuur

Na het afwerken van voorgaande stappen zijn er genoeg inzichten verworven en tools ter beschikking om een volledige architectuur te ontwerpen. Deze wordt best opgesteld door de afgelijnde subsystemen één voor één uit te werken en daarna ook de communicatie tussen deze systemen te bekijken. Denk daarbij dat een goede architectuur tot op een zeker niveau zo technologie/implementatie-onafhankelijk mogelijk moet blijven. Bouw daar waar nuttig, de nodige abstractielagen in.

Functionaliteit opsplitsen

De beste manier om alles overzichtelijk te houden is door op voorhand duidelijk de functionaliteit op te splitsen in afzonderlijke onderdelen. *Separation of concerns* is een design principe waarbij deze opsplitsing gemaakt wordt op basis van de onderdelen van de functionaliteit die op zichzelf een logisch geheel vormen (een concern).

Een eenvoudigere manier om hiermee van start te gaan, is door op zoek te gaan naar een duidelijke verantwoordelijkheid voor elke component. Als elke component een eigen (hoofd-)verantwoordelijkheid heeft (en slechts één!), dan is de verdeling veelal goed gemaakt. Aan de andere kant wordt zo onderhoud van de code ook een stuk eenvoudiger. Dankzij het duidelijk encapsuleren van bepaalde concepten en functionele stukken code, kan men eenvoudiger een stuk code en diens bedoeling begrijpen en daar waar nodig aanpassen.

Let er op dat dergelijke opsplitsing meestal ook hiërarchisch zal gebeuren. Een applicatie kan bijvoorbeeld onderverdeeld worden in een gebruikersinterface, een back-end server en een database server. De gebruikersinterface kan op zijn beurt bestaan uit een communicatie-module, een controller en dan de gebruikersinterface-elementen zelf. Nog een niveau dieper kan de communicatie-module dan weer bestaan uit bijvoorbeeld een HTTP client en een JSON parser, enzovoort.

Deze separation of concerns strategieën openen de weg naar wat men *modularisatie* van de software noemt. In modulaire software worden de verschillende componenten geëncapsuleerd in modules, die enkel communiceren met elkaar via goed afgelijnde interfaces. Modules die dezelfde interface implementeren zijn dus perfect uitwisselbaar en de verschillende modules die samen het volledige systeem vormen kunnen onafhankelijk van elkaar ontwikkeld worden. Modularisatie heeft dus als voordeelen encapsulatie, eenvoudiger onderhoud, beter begrijpbare architecturen, maar ook *information hiding* en *loose coupling* (zie volgende subsectie).

Interne details afschermen

Information hiding of dus eigenlijk het afschermen van de interne details, is een goede gewoonte bij het ontwerpen van software. Een component kan op elk niveau (van subsysteem tot enkele klasse) opgesplitst worden in enerzijds een publiek gedeelte (de interface) en anderzijds een privaat gedeelte (de interne details), hierdoor ontstaan er twee belangrijke voordelen:

- De programmeur die de component gebruikt, hoeft zich enkel aan te trekken van het publiek gedeelte en kan zich dus focussen op de eigenlijke functionaliteit die deze component aanbiedt.
- De programmeur die instaat voor de ontwikkeling van de component zelf heeft nu de mogelijkheid om de interne details van de component te wijzigen (bv. een andere datastructuur of algoritme gebruiken) zonder dat dit een impact heeft op de rest van het systeem, want het publieke gedeelte is ongewijzigd gebleven.

Op deze manier wordt er ook naar een losse koppeling (Eng.:loose coupling) tussen de componenten gestreefd. Loose coupling gaat over het maken van een zo onafhankelijk mogelijk functioneel geheel. Door het reduceren van harde afhankelijkheden, worden belangrijke software-eigenschappen zoals herbruikbaarheid, aangepasbaarheid en onderhoudbaarheid een stuk eenvoudiger realiseerbaar.

Duplicatie vermijden

Typisch kan er nu ook gemakkelijker ontdekt worden waar er eventuele duplicatie in functionaliteit zit. Als deze plaatsten geïdentificeerd worden, kan getracht worden om de functionaliteit te abstracteren tot een nieuwe component, die dan kan hergebruikt worden vanuit de andere onderdelen van het systeem.

Duplicatie in functionaliteit, duidt namelijk snel op code duplicatie en dit moet zoveel mogelijk vermeden worden. Twee goede redenen hiervoor zijn:

- **Bugs:** duplicate stukken code verhogen het risico op bugs, omdat wanneer een fout op de ééne plaats wordt opgelost, men er niet altijd direct aan denkt of gewoonweg niet weet dat deze oplossing dan ook op de overige plaatsen moet toegepast worden.
- **Onderhoud van software:** het samenbrengen van duplicate code in een abstractie heeft als voordeel dat de codebase kleiner en dus overzichtelijker wordt, maar ook dat wijzigingen veel sneller doorgevoerd kunnen worden.

21.1.5 Coderen

Eens de architectuur voldoende inzicht geeft in de werking van het systeem kan gestart worden met schrijven van de code. Een belangrijke keuze die hierbij ook gemaakt wordt, is in welke programmeertaal dit zal gebeuren. Naast de ervaring van de programmeurs

die op het project in kwestie werken, kan deze keuze ook beïnvloed worden door heel wat andere factoren, zoals bijvoorbeeld:

- **De aard van het probleem:** elke programmeertaal heeft specifieke sterktes en zwaktes, kies er één die zich goed leent voor het probleem in kwestie. Java is bijvoorbeeld goed geschikt om schaalbare en robuuste back-end infrastructuur in de cloud te ontwikkelen, terwijl C++ een goede keuze is voor applicaties waarbij uitvoeringstijd kritisch is en er dicht op de hardware gewerkt wordt (zoals bij 3D games).
- **De maturiteit van de taal:** een taal die al lang bestaat en veel gebruikt wordt, zal over betere tools en meer externe libraries beschikken. Daarnaast zal een actieve en grote community er ook voor zorgen dat je sneller oplossingen voor problemen kan terugvinden.
- **De beschikbaarheid van bestaande technologieën:** misschien zijn er bestaande pakketten die al heel wat van de vereisten voor het project inlossen, dan kan het interessant zijn om in dezelfde taal te werken om een vlotte integratie te kunnen bewerkstelligen. Vind het warm water niet opnieuw uit, er bestaan gigantisch veel kwalitatieve open-source pakketten voor allerhande probleem-domeinen!

Uit de ervaring blijkt dat een praktische manier van developen een iteratief model is, waar eerst de ontwikkelingsstappen zoals tot hiertoe beschreven gevuld worden. Eens een eerste versie van de architectuur uitgewerkt is, kan deze in code omgezet worden. Nadien kan men beginnen itereren tussen de architectuur en implementatie. Het implementeren zal namelijk onvermijdelijk bepaalde problemen aan het licht brengen, waar in de architectuur geen of slecht rekening mee gehouden was. Deze nieuwe inzichten kunnen gebruikt worden om de architectuur verder te verbeteren, waarna deze opnieuw geïmplementeerd kan worden of de gekozen technologieën bijgestuurd kunnen worden. Dit iteratief proces wordt herhaald tot zolang nodig blijkt om de vereiste kwaliteit af te kunnen leveren.

Voor verdere implementatie-tips verwijzen we naar sectie 21.2.

21.1.6 Evaluatie

In de evaluatie stap wordt de code aan een reeks van testen onderworpen om na te gaan of alle functionaliteit correct werkt. Voor grote projecten kan het ontwikkelen van dergelijke test-code soms even belangrijk zijn als de software zelf. Daarnaast is het zeker een uitdaging om alle mogelijke situaties te beschouwen (men spreekt hier dan van een bepaald niveau van *test coverage*). Doorgaans zullen er minimaal twee types van testen geïmplementeerd worden:

- **Unit testen:** Bij deze testen worden specifieke units, zoals een functie, klasse of een subsysteem in isolatie gevalueerd. De unit wordt onderworpen aan een

verzameling van gecontroleerde input data en er wordt geverifieerd of deze op de gepaste manier reageert of de correcte output genereert.

- **Integratie testen:** Bij deze testen wordt gefocused op de interactie tussen units. Een bepaald process wordt gemodelleerd waarbij verschillende units een rol spelen en er opnieuw wordt gecontroleerd of het resultaat voldoet aan de verwachtingen.

Naast een validatie-methode voor de initiële implementatie zijn dergelijke testen vaak ook de enige manier om tijdens de operationele fase te kunnen garanderen dat de software correct blijft werken. Zonder deze evaluatie stap zou een verandering aan de code (bv. een bugfix of nieuwe feature) namelijk een kettingreactie kunnen veroorzaken die ervoor zorgt dat er problemen ontstaan in andere onderdelen van de software.

21.2 Implementatie Aspecten

21.2.1 Van design tot code

De beste manier om een design om te zetten in code kan niet zomaar beschreven worden. Het is echter wel duidelijk dat de beste resultaten bekomen worden door eerst een overzicht te krijgen van wat moet gemaakt worden. Door op voorhand een duidelijke overzichtelijk architectuur op te maken, kan direct efficiënter te werk gegaan worden. Het vorige hoofdstuk zou je al goed op weg moeten zetten om een dergelijke goede architectuur op te bouwen.

Eens die architectuur er is, kan er immers direct begonnen worden met het opbouwen van de zogenaamde skeleton-code. Dit is een lege vorm van het op te leveren project, waarin alle klassen al gedefinieerd zijn, maar nog niet geïmplementeerd. Het is eigenlijk de eerste manifestatie van de architectuur in code.

Zelfs tijdens het maken van de skeleton code, kunnen al problemen aan het licht komen of mogelijke vragen opwellen. Deze kunnen direct gereflecteerd worden op de huidige architectuur en daar waar nodig aangepast worden. Eens de skeleton code er volledig is kan dan begonnen worden met stuk voor stuk componenten/klassen uit te werken, al dan niet als tijdelijke *stub* (een tijdelijke vereenvoudigde implementatie).

21.2.2 Goede coding practices

Stack versus heap

In programmeertalen waar geheugenbeheer een verantwoordelijkheid is van de programmeur, moet deze ook zelf een bewuste keuze maken wanneer en hoe al dan niet (dynamisch) geheugen gealloceerd wordt. In C/C++ hebben we het hier over stack of heap geheugen. Na het leren over (dynamische) geheugen allocatie op de heap (via malloc of new) wordt er door studenten nogal snel naar dit heap geheugen gegrepen. Ze zien deze operaties als het equivalent van de new operator in een taal als Java. Dit is echter niet

de bedoeling. Daar geheugen beheren geen simpele taak is en een bron van veel fouten (denk aan geheugen lekken), is het aangewezen om enkel zelf geheugen te alloceren op de heap, wanneer er geen andere mogelijkheid is. Voor alle andere use cases gebruikt men best het stack geheugen. Het voordeel daarvan is dat dit zelf opgeruimd worden in de code eens het *out of scope* gaat.

Heel wat studenten hebben het gevoel dat ze enkel primitiven kunnen alloceren op de stack, nochtans houdt niets hen tegen om dit met een eigen gemaakte klasse of struct te doen. Probeer dus altijd eerst te kijken in practica - en projecten in het algemeen - of het niet op te lossen valt zonder gebruik te maken van dynamische geheugen allocatie. Het zal het geheel minder complex maken en er zullen minder snel fouten optreden zoals geheugenlekken.

Memory management en verantwoordelijkheid

Een typisch terugkerende probleem is dat van *geheugenbeheer en verantwoordelijkheid*. Objecten worden gealloceerd, gebruikt en uiteindelijk moeten die terug vernietigd worden. Het grootste probleem is waar in de code deze dan vernietigd moeten worden. Dit heeft deels met stijl en overzichtelijkheid te maken, maar minstens even belangrijk is dat het geheugenlekken kan reduceren door telkens op dezelfde manier met deze kwestie om te gaan. Een goede vuistregel om te hanteren is de volgende:

De programmeur/klasse die het geheugen alloceert, is ook verantwoordelijk om dit terug vrij te geven.

In 90% van de gevallen is dit de beste methode. Daar waar de beslissing gemaakt werd om het geheugen te alloceren in de eerste plaats, wordt best ook de beslissing genomen om het terug vrij te geven. En vooral: de beslissing om het vrij te geven wordt best nergens anders genomen, want de klasse/programmeur die dit alloceerde kan onmogelijk weten dat diens geheugen reeds vrijgegeven is ergens anders.

Een typische use case is het ontwikkelen van een datastructuur. Daarbij moet ongetwijfeld een soort van *destructor* voor de datastructuur geprogrammeerd worden. Als nu de vuistregel gevolgd wordt, dan is het zo dat een andere programmeur of klasse gebruik zal maken van deze datastructuur. Dit zal gebeuren door eerst objecten aan te maken, om deze er vervolgens in te steken. Wordt nu de datastructuur verwijderd, dan mag deze niet zomaar alle objecten verwijderen die erin zitten. Het is immers perfect mogelijk dat de klasse/programmeur nog andere acties wil uitvoeren op deze objecten. Of zelfs, dat ze nog in een tweede datastructuur zitten ook, waardoor daar nu misschien *dangling pointers* in zitten.

In deze use case maakt men dus een gewone destructor die alle interne objecten en velden opruimt die nodig zijn voor het goed functioneren van de datastructuur zelf. Men zal echter niet de objecten die in de datastructuur gestoken zijn vernietigen/vrijgeven!

21.2.3 Gebruik van IDE

Debugging

Een zeer belangrijk aspect en voordeel van het gebruik van een IDE is het kunnen debuggen. Een debugger is de belangrijkst tool van een software ontwikkelaar. Het wordt, zoals het woord ook zelf aanhaalt, gebruikt om bugs op te sporen.

Al te vaak wordt een bug gezocht door gewoon wat door de code te lopen of het programma tot vervelends toe opnieuw uit te voeren. Een debugger laat ons toe om het programma te doorlopen, stap voor stap, en constant de inhoud van alle variabelen te bekijken. Als men nu de fout reproduceert, kan je perfect ontdekken waar alles verkeerd begint te gaan. Eens de bug dan opgespoord is, kan deze opgelost worden.

Het lijkt logisch, maar nog al te vaak wordt door beginnende programmeurs te veel tijd besteed zonder direct naar de debugger te grijpen.

Breakpoints

Als je een programma in debug mode opstart, zal het op het eerste gezicht niet veel anders doen, dan wanneer je het gewoon uitvoert. De bedoeling is dat je een *breakpoint* zet op de stukken in je code, waar je wilt dat het programma pauzeert, zodat je variabelen kan bekijken en stap voor stap verder kan volgen.

Bij het uitvoeren van je programma in debug mode wordt er een hoop extra informatie gegenereerd, deze informatie laten een debugger toe om exact te volgen wat er wanneer gebeurt. Het is ook op deze manier dat de debugger de uitvoering van het programma kan pauzeren, wanneer er op een breakpoint gestoten wordt.

Wat velen niet weten is dat je terwijl je programma in debug mode aan het uitvoeren is, ook gewoon breakpoints kan toevoegen. Verder zijn gewone breakpoints meestal wel gekend, maar er bestaan ook meer geavanceerdere breakpoints.

Conditionele breakpoints

Volg even het volgende voorbeeld. Stel een programma voor dat een grote lus heeft die 2000 maal doorlopen wordt. Daarbinnen blijkt dat na een 500 keer door de lus te gaan, er iets fout gaat.

Met een normaal breakpoint, zou de debugger pauzeren op de eerste keer dat de loop doorlopen wordt. De debugger kan dan de instructie krijgen om verder uit te voeren tot het volgende breakpoint, maar dat zal alweer bij de tweede iteratie van de lus zijn. Om te vermijden dat effectief 500 maal geklikt moet worden om de bug te onderzoeken, kan er in de meeste IDE's een conditioneel breakpoint aangemaakt worden. Dit is een gewoon breakpoint, met een aantal regels (if-statements als je wil). Regels van het type: *pauzeer als deze expressie waar is, pauzeer als dit breakpoint x aantal keer bereikt is, etc.* Dit is een zeer krachtig mechanisme, dat ontwikkelaars toelaat om eenvoudig complexe

bugs te traceren en op te lossen. De tijdwinst bij het oplossen van dit soort bugs, door het correct gebruiken van een debugger, is niet te onderschatten! Vergeet dit dan ook niet te gebruiken, wanneer het nuttig kan zijn.

Refactoring

Refactoring van code is een verzamelnaam voor alle *onderhoudswerken* in de code: verplaatsing, hernoeming, verwijderen, etc..

Let goed op bij hernoemen van bestanden. Veel IDE's zullen de hernoeming correct doorvoeren voor code bestanden en de inhoud van die bestanden, maar dit is niet altijd zo voor resource bestanden (eg. txt bestanden)!

Ook bij het verplaatsen van bestanden kan het zo zijn dat het een probleem geeft bij textuele referenties in resource bestanden. Altijd goed nakijken dus!

Sowieso wil je refactoring zo veel mogelijk vermijden. Dit kan je meestal doen, door op voorhand goed na te denken over de structuur en architectuur van je programma. (zie vorige hoofdstukken)

Let ook op bij het verschil tussen *remove* en *delete*. Als je uit een IDE iets verwijderd met *remove* bedoelt men ‘weg halen uit de omgeving van de IDE’, dit hoeft niet te betekenen dat het ook verwijderd wordt van de harde schijf. Een effectieve *delete* zal dan wel weer het betand van de harde schijf verwijderen.

Dit kan een belangrijk verschil zijn, wanneer bijvoorbeeld code moet opgestuurd worden voor projecten/practica. Een bestand is verwijderd uit het project met *remove* en alles werkt. Echter op het moment van indienen, wordt de volledige folder met alle source files gekopieerd en opgestuurd. Bij het openen van de code door een assistent, kan de code opeens niet meer gecompileerd worden, omdat die gewoon alle source files heeft ingeladen in de IDE.

Let hier dus goed op bij projecten en practica, maar ook in het algemeen!

21.2.4 Collaboratie

Samenwerken aan een project in software development is eerder een regel dan een uitzondering. Om efficiënt te kunnen werken zijn er heel wat tools beschikbaar. In deze sectie worden een aantal daarvan aangehaald.

Version control

Wanneer verschillende personen samen werken aan een code-file, kan dat gezien worden alsof iedereen een eigen versie van het document maakt. Deze verschillende versies

beheren is wat men *version control* noemt. Er zijn drie generaties van version control te onderscheiden:

First generation De allereerste vorm van *version control* was toen men een *lock* op een file nam en niemand anders op dat moment de file kon aanpassen. Dit is natuurlijk geen ideale manier om echt tegelijk samen te werken, maar het verhinderde wel veel problemen zoals het overschrijven van elkaars lokale aanpassingen.

Second generation De eerste modernere vorm van version control is een gecentraliseerde vorm. Er is een server waarop de code staat, dit wordt de repository genoemd. Het voordeel hiervan is dan men met meerdere tegelijk versies kan opsturen naar de server van heel het project in één keer. Om alles consistent te houden, moet wel altijd *merged* worden met de recentste bestaande versie op de server. Typische voorbeelden zijn CVS, Subversion (SVN).

Third generation De nieuwste generatie van version control is gedistribueerd. Waar bij de gecentraliseerde aanpak er altijd een centrale server nodig is (en dus in vele gevallen internet connectie), heeft iedereen nu een eigen lokale repository. Nieuwe versies post je op je eigen lokale repository. Later kan men dan beslissen om de laatste (of een andere) versie van de eigen repository te *merge*n met een versie van een remote repository op een server en deze dan uploaden (pushen). Ook vele andere workflows zijn mogelijk, zoals het binnenvullen van de nieuwe commits van de remote repository of zelfs rechtstreeks van de repository van een collega. Het heet het voordeel dat je volledig lokaal een repository kan opzetten en gebruiken en dat er veel moderne (gratis) tooling voor bestaat. Tyische voorbeelden zijn Git, Mercurial (Hg).

Issue tracking

Bij het samenwerken met meerdere wordt het al snel belangrijk om eventuele bugs te catalogiseren, zodat ze door anderen snel terug gevonden kunnen worden (en natuurlijk opgelost worden). Een ideaal systeem hiervoor is een *issue tracker*. Dit is uiteindelijk niet veel meer dan een handig te beheren lijst waar *issues* aan kunnen toegevoegd worden. Over elke van deze issues kan gediscussieerd worden, er kunnen links naar commits toegevoegd worden en commits kunnen linken naar bepaalde issues. Belangrijker misschien is dat ze ook toegewezen kunnen worden aan personen. Een aantal IDE's integreert mooi met een issue tracker, en kan een *to do list* genereren.

Issue tracker worden niet enkel gebruikt om bugs te beheren, ook features requests, enhancement voorstellen etc. kunnen hierin beheerd worden.

Documentatie

Documenteren kan op twee manieren, zowel in de code als naast het project. Beide vormen zijn belangrijk, maar documenteren vraagt ook wel behoorlijk wat werk. Probeer daarom code te documenteren daar waar dit nuttig is. Accessor en mutator methodes

(lees: *getters* en *setters*) spreken zodanig voor zich dat deze documenteren een behoorlijk tijdsverspilling is. Probeer daarentegen wel duidelijk te documenteren wat de andere publieke methodes doen, want dit zijn de methodes waar collega's mee in aanraking zullen komen. Ook grote codeblokken kunnen beter begrepen worden met een klein woordje uitleg tussen de code.

Externe documentatie die naast het project bijgehouden wordt, betreft meestal informatie over deployment en configuratie. Deze kunnen in tekstvorm of webpagina bijgehouden worden, maar worden meer en meer in een eenvoudige Wiki bijgehouden. Heel wat repository services bieden standaard een wiki sectie aan om dit soort documentatie bij te kunnen houden. (Deze wiki wordt veelal ook mee geversioned in de repository)

Extra opmerkingen

Een aantal tips bij collaboreren en versioning:

- Spreek een code-stijl conventie af: anders zal bij elke commit van iemand, elke file in het project een nieuwe versie krijgen, wegens andere styling. (bv. indentatie, accolades, etc)
- Let op met IDE-specifieke files: niet iedereen codeert in dezelfde IDE omgeving. Probeer daarom steed enkel code en project-configuratie files te versionen. IDE configuratie en metadata files versionen, kan een probleem geven voor anderen wanneer ze updaten naar de laatste versie.
- Probeer commits in meerdere kleine commits op te splitsen die voor zich spreken: dit helpt om een commit te begrijpen, maar maakt het ook makkelijker om terug te gaan naar een bepaalde versie.

github.ugent.be

Niet iedereen weet dit, maar Universiteit Gent biedt een Github Enterprise service aan voor zijn studenten (zie ook figuur 21.2). Op <https://github.ugent.be> kan elke student inloggen met zijn gent-authenticatie gegevens (zelfde als op minerva). Iedereen kan er een nieuw project aan maken met:

- Git code repository
- Issue tracker
- Wiki
- Statistieken
- Mogelijkheid om heel wat services te koppelen (e.g. build processen)

Hoofdstuk 21: Software ontwikkeling in de praktijk

The screenshot shows the GitHub homepage with the 'GitHub Bootcamp' repository selected. The repository page features a 'News Feed' tab, which is currently active, showing four cards: 'Set up Git', 'Create repositories', 'Fork repositories', and 'Work together'. Below the news feed, there is a list of recent commits:

- 18 days ago: [whenshine pushed to develop at libeun-intranet](#)
fbc1cc Implemented notifications when adding project members
- 19 days ago: [whenshine pushed to develop at libeun-intranet](#)
c34f06 Working mail notifications for project access requests, by adding ...
462fa42 Fixed bug when adding project as a user that is also the item lea...
[View comparison for these 2 commits >](#)
- 19 days ago: [whenshine pushed to develop at libeun-intranet](#)
46bd802 Fixed some bugs with project member assignments and request ...
0609882 Deleted deleted delete remnant in DAO
[View comparison for these 2 commits >](#)

On the right side of the repository page, there is a sidebar titled 'Your repositories (3)' with a 'New repository' button and a search bar. It lists three repositories: 'libeun-intranet', 'whenshine/traptest', and 'whenshine/repo-test'.

Figuur 21.2: github.ugent.be

Let wel goed op als je deze service gebruikt en je maakt een nieuwe repository aan voor een project of practicum: **altijd private kiezen!** Publieke repositories kunnen namelijk door iedereen gezien worden en dus ook afgekeken worden door andere groepen/studenten.

Tools

Naast de command-line git client, bestaan er ook heel wat grafische client tools. Hieronder staan twee gratis clients met hun referentie, maar er bestaan uiteraard nog veel anderen:

- SourceTree (Windows & Mac): <http://sourcetreeapp.com>
- GitEye (Windows, Mac & Linux): <http://www.collab.net/giteyeapp>

Dikwijls steunen dit soort grafische tools op een specifieke merge tool om twee commits te mergen. Deze moet soms apart geïnstalleerd worden en aangewezen worden in de configuratie instellingen. (e.g. KDiff3 - <http://kdiff3.sourceforge.net>)

Deel V

Software Ontwikkeling in Objective-C

Hoofdstuk 22

Objective-C

22.1 Korte historiek

De taal Objective-C werd gecreëerd door Brad Cox en Tom Love begin de jaren '80 in hun softwarebedrijf Stepstone. Beiden hadden een Smalltalk achtergrond en waren sterk overtuigd van de Smalltalk principes en van noodzakelijkheid van compatibiliteit met de programmeertaal C. Objective-C is een object-georiënteerde uitbreiding van C waarbij de Smalltalk visie en syntax van object-oriëntatie gehanteerd wordt. Objective-C is een superset van de programmeertaal C: elk C programma kan met een Objective-C compiler gecompileerd worden en C code kan steeds gebruikt worden voor de implementatie van methoden van Objective-C klassen.

Het bedrijf NeXT van Steve Jobs kocht in 1988 een licentie op Objective-C van Stepstone en breidde de `gcc` compiler uit met Objective-C ondersteuning. Hun voornaamste product was NeXTStep, een ontwikkelomgeving voor object-georiënteerde applicaties op hun besturingssysteem. Dit product was zeer populair begin de jaren '90.

Objective-C is sindsdien de native taal voor software ontwikkeling op het Mac OS X besturingssysteem en ook voor het iOS besturingssysteem, gebruikt op Apple toestellen zoals de iPhone, de iPod Touch, en de iPad. Cocoa is de Objective-C application programming interface (API) voor het Mac OS X besturingssysteem en Cocoa Touch is de Objective-C API voor de Apple mobiele toestellen.

Zoals vermeld is de Objective-C syntax geïnspireerd op Smalltalk en de niet-object-georiënteerde operaties (zoals primitieve datatypes, pre-processing, expressies, functie declaraties, en functie oproepen) zijn identiek aan deze van C. Net zoals in C spelen ook pointers en geheugenbeheer een belangrijke rol in Objective-C.

22.2 Boodschappen - Eng.:Messages

Het Objective-C model van object-georiënteerd programmeren is gebaseerd op het versturen van boodschappen (Eng.: messages) naar object instanties. In plaats van een methode op te roepen op een object, wordt een boodschap gestuurd naar het object.

In tegenstelling tot C++ wordt at runtime bepaald naar welk object de boodschap gestuurd wordt en het ontvangend object interpreteert dan de boodschap (de compiler kan wel een waarschuwing geven wanneer een niet gedefinieerde boodschap naar een object instantie gestuurd wordt). Wanneer een object een boodschap niet herkent wordt at runtime een exceptie gegenereerd. Een eigenschap een programeertaal die gebaseerd is op het uitwisselen van boodschappen (zoals Smalltalk en Objective-C) is dat er geen type checking tijdens compilatie plaatsvindt, maar tijdens de uitvoering van het programma. Het gevolg is uiteraard dat de uitvoering minder snel gaat, maar dat er wel meer flexibiliteit tijdens de uitvoering is.

De Objective-C syntax om een methode van een object op te roepen, is als volgt:

```
[object_ptr method:arg_value];
```

waarbij `object_ptr` een pointer naar een object (aangemaakt op de heap) voorstelt, `method` de naam van de methode in de klasse gedeclareerd en gedefinieerd, en `arg_value` de waarde die als argument doorgegeven wordt.

Deze oproep is equivalent met de volgende invokatie-syntax van C++:

```
object_ptr->method(arg_value);
```

In geval meerdere argumenten worden doorgegeven is de syntax in C++ uiteraard:

```
object_ptr->method(arg_value1, arg_value2, arg_value3);
```

en in Objective-C zijn er twee mogelijkheden, namelijk ofwel:

```
[object_ptr method:arg_value1:arg_value2:arg_value3];
```

ofwel:

```
[object_ptr method:arg_value1 arg2_descr:arg_value2 arg3_descr:arg_value3];
```

waarbij `arg2_descr` en `arg3_descr` een tekstuele beschrijving zijn (in één woord) van respectievelijk het tweede en derde argument. Een dergelijke tekstuele beschrijving is handig bij bijvoorbeeld oproep van API methoden met meervoudige argumenten.

Het versturen van een boodschap wordt altijd tussen vierkante haakjes ([en]) in de code vermeld. Er zijn drie delen tussen de haakjes:

1. *ontvanger*: een pointer naar het object dat geacht wordt om de methode uit te voeren,
2. *selector*: de naam van de uit te voeren methode,
3. *argumentwaarden*: de waarden die als parameters aan de methode doorgegeven worden.

22.3 Interfaces en Implementaties

In Objective-C worden de declaraties van klassen in headerbestanden geplaatst, net zoals in C met de `*.h` extensie. De implementatiebestanden, die de implementatie code van de gedeclareerde methoden bevatten, krijgen de extensie `*.m` (m van *messages*). Een headerbestand bevat een interface-beschrijving van een klasse en de implementatie ervan komt, net als in C, in een apart bestand. De naamgeving van de bestanden kan vrij gekozen worden, maar het is wel handig als de naam van het headerbestand verwijst naar de klassenaam en het implementatiebestand een vergelijkbare (ofzelfde) naam heeft als het headerbestand.

22.3.1 Interface

Een voorbeeld van een interface declaratie in een headerbestand (`A.h`) is hieronder weergegeven:

```
#import <Foundation/Foundation.h>

@interface A : NSObject
{
    NSString *str;
    NSDate *date;
    int x;
}

// Getters
-(NSString *) str;
-(NSDate *) date;
-(int) x;

// Setters
-(void) setStr:(NSString *)input;
-(void) setDate:(NSDate *)input;
-(void) setX:(int)input;
-(void) setStr:(NSString *)str andDate:(NSDate *)date andInteger:(int)x;
-(void) setAttributes:(NSString *)str:(NSDate *)date:(int)x;

// Initializers
-(id) init;
-(id) initStr:(NSString *)str andDate:(NSDate *)date andInteger:(int)x;

// Instance Methods
-(void) printInstanceVars;
-(void) printInstanceVars:(id)input, ...;
-(void) dealloc;

// Class Methods
+(id) randomObject;
```

```
@end
```

De klasse **A** bevat 3 attributen: **str**, **date** en **x**. In Objective-C worden objecten steeds op de heap bijgehouden en bijgevolg wordt steeds met pointers naar de objecten gewerkt. De basisklasse waarvan de klasse **A** overerft is de klasse **NSObject**, de gemeenschappelijke basisklasse van alle objecten (zelfde principe als in Java). De prefix **NS** is een historisch overblijfsel van de hierboven vermelde NeXTStep (of NS afgekort). Het **Foundation** raamwerk (Eng.: framework) wordt steeds gebruikt voor de basis methoden en klassen. Men onderscheidt 4 types van methoden:

- Accessor Methoden - Eng.:Accessor Methods: deze methoden zorgen voor toegang tot de attributen. Er wordt verder onderscheid gemaakt tussen get-methoden (ook getters genoemd) voor leestoegang tot de attributen en set-methoden (ook setters genoemd) voor schrijftoegang tot de attributen. De accessor methoden zorgen ervoor dat men vanuit externe objecten toegang kan krijgen tot de attributen (zonder deze methoden zijn de attributen enkel zichtbaar binnen de klasse zelf). De methoden **setStr** en **setAttributes** illustreren meervoudige argumenten bij methode-declaraties.
- Initialisatie Methoden - Eng.:Initializers: vermits Objective-C geen constructoren gebruikt, dienen objecten geïnitialiseerd te worden door een initialisatie-methode op te roepen. Elke initialisatie-methode begint met het sleutelwoord **init** (naam conventie). In het voorbeeld hierboven worden twee initialisatie-methoden aan de klasse **A** toegevoegd. Initialisatie-methoden hebben steeds **id** als return type, welke overeenkomt met **void*** in C en C++. De reden waarom **id** als return type gebruikt wordt en niet **A** of **NSObject** is omdat men anders in een afgeleide klasse met ook een **init** methode, het probleem heeft dat dezelfde methodenaam gebruikt wordt maar met een verschillend return type: dit is net als in C of C++ niet toegelaten in Objective-C.
- Instantie Methoden - Eng.:Instance Methods: dit zijn de eigenlijke methoden die de klasse aanbiedt, zoals de **printInstanceVars** en **dealloc** methoden in het voorbeeld hierboven. De tweede **printInstanceVars** methode is een voorbeeld van een methode met variabel aantal argumenten (gebruik van ellipsis ...).
- Klasse methoden - Eng.:Class Methods: de methoden hierboven beschreven werken op objecten van een klasse (instanties van een klasse) en worden in het headerbestand steeds vooraf gegaan door een - teken. Klasse methoden worden opgeroepen op de klasse zelf en worden om het onderscheid te maken in het headerbestand steeds vooraf gegaan door een + teken. De methode **randomObject** is een voorbeeld: deze creëert een object, vult random waarden in en geeft dit aangemaakte object terug. Klasse methoden hebben geen rechtstreekse toegang tot de attributen (enkel via de accessor methoden).

22.3.2 Implementatie

De onderstaande code toont de implementatie van de klasse A hierboven (deze implementatie komt in het bestand `A.m`):

```
#import "A.h"

@implementation A

// =====
// = Accesssor methoden =
// =====

-(int) x
{
    return x;
}

-(NSString *) str
{
    return str;
}

-(NSDate *) date
{
    return date;
}

-(void) setStr:(NSString *)inputStr
{
    [inputStr retain];
    [str release];
    str = inputStr;
}

-(void) setDate:(NSDate *)input
{
    [input retain];
    [date release];
    date = input;
}

-(void) setX:(int)input
{
    x = input;
}

-(void) setStr:(NSString *)strInput
           andDate:(NSDate *)dateInput
```

```

        andInteger:(int)xInput
{
    [self setStr:strInput];
    [self setDate:dateInput];
    [self setX:xInput];
}

-(void) setAttributes:(NSString *)strInput:(NSDate *)dateInput:(int)xInput
{
    [self setStr:strInput];
    [self setDate:dateInput];
    [self setX:xInput];
}

// =====
// = Initialisatie Methoden =
// =====

-(id) init
{
    [super init];
    [self setStr:@""];
    [self setDate:nil];
    [self setX:999];
    return self;
}

-(id) initStr:(NSString *)strInput
    andDate:(NSDate *)dateInput
    andInteger:(int)xInput
{
    [super init];
    [self setStr:strInput];
    [self setDate:dateInput];
    [self setX:xInput];
    return self;
}

// =====
// = Instantie Methoden =
// =====

-(void) printInstanceVars
{
    // direct access to the instance variables
    NSLog(@"%@", x, str, date);
}

-(void) printInstanceVars:(id)input, ...

```

```

{
    id currentObject;
    va_list argList;
    int objectCount = 1;

    if (input)
    {
        NSLog(@"%@", Object #%d is: %@", objectCount++, input);

        va_start(argList, input);
        while (currentObject = va_arg(argList, id))
            NSLog(@"%@", Object #%d is: %@", objectCount++, currentObject);
        va_end(argList);
    }
}

-(void) dealloc
{
    // no release needed of the integer instance variable x
    [str release];
    [date release];
    [super dealloc];
}

// =====
// = Klass Methoden =
// =====

+(id) randomObject
{
    return [[self alloc] initStr:@"random" andDate:nil andInteger:rand()];
}
@end

```

Deze code illustreert de volgende belangrijke principes van Objective-C:

1. **#import** zorgt voor includeren van bestanden door de preprocessor (i.p.v. **#include** in C en C++)
2. de methoden **setStr** en **setDate** maken gebruik van geheugenbeheer (de boodschappen **retain** en **release** komen verder in dit hoofdstuk aan bod).
3. **self** houdt het adres bij van het huidige object, volledige vergelijkbaar met **this*** in C++ en **this** in Java
4. **super** bevat het adres van de bovenliggende klasse (basisklasse) en laat toe om boodschappen uit de basisklasse te sturen naar het object

5. de klasse **NSString** is de standaard string-klasse uit de **Foundation** bibliotheek: het is net als de `std::string` klasse in C++ een volwaardige klasse met zeer veel gedefinieerde methoden.
6. constante strings hebben steeds de vorm `@"..."`
7. de **init** geeft het geïnitialiseerde object zelf terug als return waarde, zodat het in uitdrukkingen onmiddellijk kan gebruikt worden
8. **nil** is de nul-pointer en komt overeen met `NULL` in C en C++
9. de **NSLog** functie (geen methode) zorgt voor console output, vergelijkbaar met de **printf** functie in C. Er wordt ook met een formaat string gewerkt, die conversiekarakters bevat, en waarin de argumenten worden gesubstitueerd. De conversiekarakters zijn identiek aan deze in C, behalve het `%@` conversiekarakter wordt gebruikt voor output van een object (bijvoorbeeld voor output van een **NSString** object, en dus vergelijkbaar met `%s` in C)
10. de implementatie van de tweede methode `printInstanceVars` illustreert de implementatie van methoden met variabele argumenten (volledig vergelijkbaar met de implementatie in C en C++)
11. **dealloc** is een belangrijke methode voor geheugenbeheer in Objective-C en komt verder aan bod in dit hoofdstuk
12. de **randomObject** methode is een voorbeeld van een klasse methode, die een nieuw object aanmaakt en teruggeeft als return waarde.

22.3.3 Aanmaken van objecten

Dit gebeurt in Objective-C door eerst een ongeïnitialiseerde instantie van een klasse te alloceren (via de **alloc** methode) en dit vervolgens te initialiseren. Deze beide stappen gebeuren typisch in dezelfde regel, zoals hieronder geïllustreerd:

```
A *ptr = [[A alloc] init];
A *ptr = [[A alloc] initStr:@"----" andDate:[NSDate date] andInteger:97];
```

De **alloc** methode is een methode uit de basisklasse **NSObject** en de initialisatiemethoden **init** en **initstr** werden in het voorbeeld hierboven gedeclareerd en geïmplementeerd.

22.3.4 Vernietigen van objecten

Om een object te verwijderen dient de **release** boodschap naar dit object gestuurd te worden. Om de variabele **ptr** uit het voorbeeld hierboven weer vrij te geven, is dus volgende regel nodig:

```
[ptr release];
```

Uiteraard is `ptr` na deze operatie een zwevende wijzer (Eng.: dangling pointer). Het is veiliger om de pointer ook op nul te zetten:

```
ptr=nil;
```

In Objective-C wordt de nul-wijzer aangeduid door `nil` (i.p.v. `NULL` in C of C++).

22.4 Compileren en Linken

Hetzelfde volledig zelfde principe als in C en C++ wordt gehanteerd: bronbestanden (met extensie `*.m`) worden één voor één omgezet naar objectbestanden (met extensie `*.o`) en vervolgens worden alle objectbestanden gelinkt tot één uitvoerbaar bestand. Compileren van Objective-C kan met de GNU gcc compiler en is op Windows beschikbaar door installatie van **GNUStep MSYS**, **GNUStep Core** en de **GNUStep Devlloper** package. GNUStep is een open source Cocoa project.

Het makebestand om de applicatie hierboven (bestanden `A.m`, `A.h` en `main.m`) te compileren en te linken is bijvoorbeeld:

```
INCLUDE_PATH = -I/c/GNUstep/GNUstep/System/Library/Headers
LIBRARY_PATH = -L/c/GNUstep/GNUstep/System/Library/Libraries
LFLAGS = -lobjc -lgnustep-base
FLAGS = -fconstant-string-class=NSConstantString -Wno-import
CC = gcc

all: A.exe

A.exe: A.o main.o
$(CC) -o $@ $^ $(LIBRARY_PATH) $(LFLAGS)

A.o: A.m A.h
$(CC) -c $< $(INCLUDE_PATH) $(FLAGS)

main.o: main.m
$(CC) -c $< $(INCLUDE_PATH) $(FLAGS)
```

Het commando:

```
make all
```

of:

```
make A.exe
```

zorgt dan voor het aanmaken van het gewenste uitvoerbaar bestand.

In Objective-C kan een boodschap naar een object gestuurd worden dat niet in de interface is gedeclareerd: dit principe wordt *dynamic typing* genoemd: de exacte types hoeven tijdens het compileren niet perfect overeen te komen, maar tijdens de uitvoering wordt nagegaan of een object in staat is om zinvol te antwoorden op een boodschap. Een

object kan een boodschap ook doorsturen naar een ander object, die de boodschap wel kan beantwoorden. Indien een boodschap gestuurd wordt naar een object met adres `nil`, dan wordt dit ofwel genegeerd tijdens de uitvoering ofwel wordt een exceptie gegenereerd (afhankelijk van de compiler opties).

22.5 Protocollen - Eng.:Protocols

In Objective-C kan je een protocol definiëren: een lijst van methoden die een klasse volledig of gedeeltelijk kan implementeren. Dit is vergelijkbaar met het `interface` principe in Java en C#. Er wordt onderscheid gemaakt tussen een ad hoc protocol, ook informeel protocol (Eng.: informal protocol) genoemd (waarbij niet alle methoden van het protocol dienen voorzien te worden) en een verplicht protocol, ook formeel protocol (Eng.: formal protocol) genoemd (waarbij de compiler afdwingt dat alle methoden van het protocol voorzien worden).

Beschouwen we volgend voorbeeld van een protocol declaratie (in een `*.h` bestand):

```
@protocol FileIO
- (void)writeToFile:(NSString*)fileName;
- (void)readFromFile:(NSString*)fileName;
@end
```

Wanneer de klasse A, hierboven beschreven, dit protocol implementeert is de syntax van de declaratie (in het bestand `A.h`) als volgt:

```
@interface A : NSObject <FileIO>
{
    NSString *str;
    NSDate *date;
    int x;
}
...
// Protocol Methods
- (void)writeToFile:(NSString*)fileName;
- (void)readFromFile:(NSString*)fileName;
@end
```

en de implementatie van de klasse (in het bestand `A.m`):

```
@implementation A
...
// =====
// = Protocol Methods =
// =====
- (void)writeToFile:(NSString*)fileName
{
...
}
```

```

}
- (void)readFromFile:(NSString*)fileName
{
    ...
}
@end

```

Een standaard compiler geeft waarschuwingen als er methoden uit het protocol niet gedeclareerd en/of niet geïmplementeerd zijn, maar genereert wel uitvoerbare code.

22.6 Doorsturen - Eng.:Forwarding

Zoals vermeld laat Objective-C toe dat boodschappen gestuurd worden naar een object dat niet kan antwoorden op een boodschap (omdat de corresponderende methode niet aanwezig is). Het object kan de boodschap gewoon laten vallen of kan het doorsturen naar een object dat wel kan antwoorden. Dit principe van doorsturen (Eng.: forwarding) kan gebruikt worden om een implementatie te vereenvoudigen of om een ontwerpspatroon (Eng.: design pattern), zoals het Observer ontwerpspatroon of Proxy ontwerpspatroon, te implementeren.

22.7 Categorieën - Eng.:Categories

Een categorie laat toe om de implementatie van methoden over verschillende bestanden te spreiden: men groepeert gerelateerde methoden in categorieën om de code meer leesbaar te maken.

Bovendien worden de methoden binnen een categorie tijdens de uitvoering aan een klasse toegevoegd. Dit laat toe dat methoden aan een bestaande (en reeds compileerde) klasse kunnen toegevoegd worden zonder de klasse opnieuw te compileren of de broncode aan te passen.

Methoden in een categorie kunnen tijdens de uitvoering niet onderscheiden worden van andere methoden. Een methode in een categorie heeft ook volledige toegang tot alle attributen van de klasse waar de categorie bijhoort.

De code hieronder toont een voorbeeld van een categorie met naam `formatString` om aan de bestaande `NSString` klasse een methode toe te voegen (in het bestand `formatString.h`):

```

#import <Foundation/NSString.h>

@interface NSString (formatString)
+(NSString *) addDashes: (NSString *)inputString;
@end

@interface NSMutableString (formatString)
-(NSMutableString *) addDashes;
@end

```

De toegevoegde methode voegt de string "–" voor en achteraan een bestaande string toe. Bemerk dat we aan de `NSString` klasse een klasse methode toevoegen (met `+ teken`) en aan de `NSMutableString` klasse een instantie-methode. `NSMutableString` is een afgeleide klasse van de `NSString` klasse, die ook aanpassing van de interne string toelaat. Dit is vergelijkbaar met de `NSArray` en `NSMutableArray` klassen in Objective-C. De code hieronder toont de implementatie van de twee methoden (in het bestand `formatString.m`):

```
#import "formatString.h"

@implementation NSString (formatString)
+(NSString *) addDashes: (NSString *)inputString
{
    return [NSString stringWithFormat:@"--%@",inputString];
}
@end

@implementation NSMutableString (formatString)
-(NSMutableString *) addDashes
{
    [self insertString:@"--" atIndex:0];
    [self appendString:@"--"];
    return self;
}
@end
```

Het gebruik van deze code (in bijvoorbeeld het bestand `main.m`) wordt hieronder geïllustreerd:

```
 NSLog(@"%@", [NSString addDashes:@"sample string"]);

NSMutableString* str = [[NSMutableString alloc] initWithString:@"ingevulde string"];
NSLog(@"%@",[str addDashes]);
NSLog(@"%@",str);
```

Uitvoering van deze code geeft als output:

```
--sample string--
--ingevulde string--
--ingevulde string--
```

In geval een categorie een methode declareert met dezelfde signatuur als een bestaande methode, dan krijgt de categorie methode steeds prioriteit. Dit is dus een krachtig principe om methoden toe te voegen aan een bestaande klasse, maar ook om de implementatie van methoden te vervangen. Dit kan bijvoorbeeld gebruikt worden om bugs op te lossen, zonder de source code te moeten aanpassen.

22.8 Overerving in Objective-C

Overerving in Objective-C is vergelijkbaar met overerving in C++. Na de klassenaam in een declaratie vermeldt men na : de basisklasse. Er kan echter maar één basisklasse zijn (net zoals in Java).

22.8.1 Initialisatie van objecten van afgeleide klassen

Vermits er geen constructoren in Objective-C gebruikt worden, maar vervangen zijn door initialisatie-methoden, dient in de initialisatie-methoden speciale aandacht besteed te worden aan de expliciete oproep van de initialisatie-methoden van de bovenliggende klasse (door gebruik te maken van het `super` adres). De code hieronder illustreert dit (uit het `A.m` bestand uit sectie 22.3.2):

```
- (id)initWithValues:(NSString *)name (NSDate *)date (int)value
{
    [super init];
    [self setStr:name];
    [self setDate:date];
    [self setX:value];
}
```

Dit is volledig vergelijkbaar met de expliciete oproep van de constructor van de bovenliggende klasse in C++ (in de initialisatie-sectie van de constructor).

22.8.2 Verwijdering van objecten van afgeleide klassen

In C++ worden destructoren van bovenliggende klassen impliciet opgeroepen. In Objective-C daarentegen dient expliciet de `dealloc` methode van de bovenliggende klasse opgeroepen te worden nadat de attributen van de klasse zelf zijn vrijgegeven. De code hieronder illustreert dit (voor de `dealloc` methode van de klasse A uit sectie 22.3.2):

```
- (void)dealloc
{
    [str release];
    [date release];
    [super dealloc];
}
```

22.8.3 isa

Elk object in Objective-C heeft een attribuut met als naam: `isa`. Deze bevat het adres van de klasse die het object heeft aangemaakt. Op deze manier kan men te weten komen van welke afgeleide klasse een object een instantie is. Dit is volledig vergelijkbaar met `instanceof` in Java of het dynamic casting principe in C++.

22.9 Verschillen met C++

Hieronder worden een aantal van de belangrijkste reeds in dit hoofdstuk behandelde verschillen met C++ samengevat en worden ook extra verschillen aangebracht:

1. allocatie en initialisatie methoden in plaats van constructoren in C++ en deallocatie methoden in plaats van destructoren in C++: deze dienen steeds expliciet opgeroepen te worden

2. in Objective-C worden objecten steeds op de heap aangemaakt, nooit op de stack.
Er wordt dus steeds met pointers naar objecten in de code gewerkt
3. geen templates in Objective-C: als alternatief zijn er collector klassen (zoals `NSArray` en `NSMutableArray`) die elementen van het type `NSObject` bevatten. Instanties van collector klassen kunnen dus heterogene objecten (i.e. van verschillende types) bevatten
4. de overerving in Objective-C is enkelvoudig: een klasse kan slechts 1 basisklasse hebben in tegenstelling tot overerving in C++
5. er is geen operator overloading in Objective-C, in tegenstelling tot C++
6. de `#include` preprocessor directieve in C en C++ is vervangen door `#import` in Objective-C
7. er zijn geen referentie types (met `&` syntax) zoals in C++ (voor doorgeven van call-by-reference argumenten)
8. in C++ wordt het type `bool` gebruikt met mogelijke waarden `true` en `false`, in Objective-C wordt het `BOOL` type gebruikt met mogelijke waarden `YES` en `NO`
9. `void*` en `NULL` in C++, zijn vervangen door respectievelijk `id` en `nil` in Objective-C
10. in Objective-C kunnen boodschappen naar `nil` gestuurd worden, terwijl in C++ het oproepen van een methode op een object met adres `NULL` als gevolg heeft dat het programma zal crashen (wgens geheugenfout)
11. Objective-C laat toe dat accessor methoden automatisch door de compiler gegenereerd worden door gebruikt te maken van de `@property` uitdrukking (zoals in de volgende sectie aan bod komt).
12. Objective-C kent niet het principe van naamruimten (Eng.: name spaces) in C++, net zoals C.

In de volgende sectie wordt dieper ingegaan op het belangrijke onderwerp van geheugenbeheer in Objective-C.

22.10 Geheugenbeheer in Objective-C

22.10.1 Verschillende opties

Er zijn drie opties voor geheugenbeheer in Objective-C:

1. MRR (Eng.: *Manual Retain-Release*): de programmeur alloceert manueel geheugen en geeft dit vrij als het niet meer nodig is, dus expliciet geheugenbeheer door de applicatieontwikkelaar. De benaming Retain-Release komt voort van het feit dat

de programmeur `retain` en `release` boodschappen stuurt voor manueel geheugenbeheer. Deze boodschappen komen verder aan bod in sectie 22.10.4. Er wordt gebruik gemaakt van reference counting: er wordt bijgehouden per object hoeveel andere objecten dit object nodig hebben. Reference counting wordt voorzien in de `Foundation/NSObject` klasse in interactie met de run time omgeving.

2. ARR (Eng.: *Automatic Reference Counting*): de boodschappen voor geheugenbeheer dienen niet door de applicatieontwikkelaar voorzien te worden, maar de compiler voegt deze automatisch toe tijdens het compilatie-proces. Er wordt op dezelfde manier als bij MRR gebruik gemaakt van reference counting. Echter nu worden de reference counts automatisch aangepast (door code die door de compiler toegevoegd wordt). ARR wordt soms verward met garbage collection (3e optie voor geheugenbeheer in Objective-C): bij garbage collection is er echter een achtergrond proces at runtime dat controleert of objecten nog wel gebruikt worden, en indien niet worden deze objecten opgeruimd. Bij reference counting is er geen dergelijk achtergrond proces en wordt een object met een reference count van waarde 0 onmiddellijk opgeruimd.
3. AGC (Eng.: *Automatic Garbage Collection*): volledig vergelijkbaar met geheugenbeheer in Java, de runtime omgeving houdt bij welke objecten nog gereferereerd worden en objecten die niet meer nodig zijn, worden door een achtergrond proces regelmatig opgeruimd. Deze optie is enkel aanwezig in Objective-C 2.0 en de applicatieontwikkelaar heeft de keuze om dit al dan niet te gebruiken (geen verplichting zoals in Java).

Vermits voor iPhone Cocoa Touch applicaties de eerste optie (MMR) noodzakelijk is, wordt deze verder in deze sectie behandeld. Indien een platform ARC ondersteunt, is dit de meest aangewezen aanpak. Bij gebruik van ARC is de verdere informatie in dit hoofdstuk niet nodig om applicaties te ontwikkelen, maar het verschafft wel verder inzicht. Vermits momenteel garbage collection geen optie is voor iPhone applicaties, is het zeker belangrijk om de onderstaande concepten (reference counts, object ownership, etc.) te begrijpen.

Wanneer `gcc` als Objective-C compiler gebruikt wordt, kan men volgende command line opties meegeven:

- `-fobjc-no-arc` : geeft aan dat men niet van ARC maar wel van MMR wil gebruiken.
- `-fobjc-arc` : geeft aan dat men wel van ARC wil gebruiken.
- `-fobjc-gc` : gebruik van AGC.

22.10.2 Bezit van een object - Eng.: Object Ownership

Als in een object een ander object aangemaakt wordt, dan is het eerste object de eigenaar (Eng.: owner) van het tweede object. Als dit aangemaakte object dan doorgegeven

wordt aan een derde object, kan dit derde object er dan voor opteren om ook eigenaar te worden: de reference count van het object is dan 2 (m.a.w. er zijn twee eigenaars). Als één van de eigenaars het object niet meer nodig heeft en afstaat, wordt de reference count van het object dan 1 (m.a.w. er is slechts één eigenaar meer). Wanneer een object geen eigenaars meer heeft, wordt het in de MRR en ARC gevallen *onmiddellijk* verwijderd.

In het MMR model zijn de eigenaars verantwoordelijk om aan te geven wanneer ze een object niet meer nodig hebben. Dan wordt de reference count met één verminderd. Als de reference count van een object op nul valt, dan wordt het ogenblikkelijk verwijderd. Indien men een object wil gebruiken, dan kan dit uiteraard ook zonder dat men de eigenaar is van het object: men heeft de garantie dat het in een methode oproep niet verwijderd wordt, maar het kan in principe na de methode oproep onmiddellijk verwijderd worden. Indien de ontwikkelaar dus 100% zeker wil zijn dat een object blijft bestaan gedurende een langere tijd, dient hij eigenaar te worden van het betreffende object.

22.10.3 Aanmaken van objecten via klasse methoden en via instantie methoden

Zoals eerder in dit hoofdstuk aangegeven zijn er twee manieren om een object aan te maken:

1. via de `alloc` klasse methode en vervolgens een initialisatieboodschap te versturen.
De code hieronder illustreert dit:

```
NSString *str1 = [[NSString alloc] initWithFormat:@"%@", @"initialisatiestring"];
```

dit is een expliciete allocatie, het object waarin deze allocatie plaatsvindt wordt de eigenaar en dient een `release` boodschap te versturen wanneer `str1` niet meer nodig is.

2. onmiddellijk via een klasse methode, zoals hieronder geïllustreerd:

```
NSString *str2 = [NSString stringWithFormat:@"%@", @"initialisatiestring"];
```

in dit geval is er geen expliciete allocatie en het object waarin deze allocatie plaatsvindt wordt geen eigenaar en dient dus geen `release` boodschap te versturen om `str1` vrij te geven. Het object komt terecht in de zogenaamde `autorelease` pool en verwijderd wanneer deze vrijgegeven wordt.

22.10.4 Boodschappen voor geheugenbeheer

Volgende boodschappen zijn belangrijk voor manueel geheugenbeheer:

- `alloc` (bijv. `[NSString alloc]`): alloceert een instantie van een klasse en zet de reference count op 1. Het object waarin deze allocatie verstuurd wordt, wordt de eigenaar en dient het object na gebruik weer vrij te geven.

- **new** (bijv. `[NSString new]`): dit is een verkorte notatie van `[[NSString alloc] init]`.
- **retain** (bijv. `[str1 retain]`): dit geeft aan dat men van een bestaand object, dat elders aangemaakt werd, ook eigenaar wil worden. De reference count wordt met 1 vermeerderd en het object krijgt er een nieuwe eigenaar bij. Het object kan pas vrijgegeven worden, als het door alle eigenaars is vrijgegeven.
- **release** (bijv. `[str1 release]`): dit geeft aan dat een object niet langer nodig is, het eigenaarschap wordt opgegeven en de reference count wordt met 1 verminderd. Als de reference count hierdoor op 0 valt, wordt het object direct verwijderd en het ingenomen geheugen wordt vrijgegeven. Belangrijk is uiteraard dat enkel eigenaars **release** boodschappen op hun objecten mogen versturen. De compiler controleert dit niet, maar kan voor run-time onaangename verrassingen zorgen.
- **autorelease** (e.g. `[str1 autorelease]`): dit geeft aan dat het object tijdelijk nog nodig is, maar dat je niet een eigenaar van het object wilt worden. Dit komt aan bod in de sectie 22.10.6 hieronder over autorelease pools.
- **copy** (e.g. `str2 = [str1 copy]`): dit zorgt voor een diepe kopie van een object (indien uiteraard de **copy** methode correct is geïmplementeerd) en het object waarin deze kopie operatie verstuurd wordt, wordt de eigenaar van het nieuwe object.

22.10.5 Geheugenbeheer: vuistregels

Het is belangrijk om steeds de juiste van de bovenstaande boodschappen te versturen bij manueel geheugenbeheer (MRR). Er zijn twee belangrijke vuistregels:

1. Als je een object bezit (via **alloc**, **retain** of **copy** boodschappen), dan dien je dit ook vrij te geven (door zelf expliciet een **release** boodschap naar het object in kwestie te versturen).
2. Als je een object niet bezit (aangemaakt via een klasse methode, zoals in sectie 22.10.3 aangegeven of via een argument doorgegeven), dan mag je het object NIET vrijgeven (geen **release** boodschap naar het object in kwestie versturen).

22.10.6 Autorelease pool

Een autorelease pool is een instantie van de klasse `NSAutoreleasePool` en houdt tijdelijke objecten bij die automatisch moeten vrijgegeven worden (vandaar de naam **autorelease**). Er zijn twee types objecten die in de autorelease pool terechtkomen:

1. objecten waarnaar expliciet de **autorelease** boodschap is verstuurd,
2. objecten aangemaakt via een klasse methode, zoals in sectie 22.10.3 aangegeven.

Als de autorelease pool wordt vrijgegeven (tijdens de uitvoering van een programma) worden alle objecten in de autorelease pool automatisch vrijgegeven. Dit is een eenvoudige manier van geheugenbeheer voor tijdelijke objecten.

22.10.7 Geheugenbeheer bij gebruik van containers

De basisregel is dat arrays, dictionaries of andere containers eigenaar worden van objecten die in de container gestockeerd worden. Wanneer een object uit een container verwijderd wordt, is de container niet langer eigenaar (reference count wordt met 1 verminderd). Als een container vrijgegeven wordt, wordt naar alle elementen in de container een **release** boodschap gestuurd.

Wanneer men container gebruikt, dient men voor alle zekerheid te controleren (bijv. in de documentatie) of de container wel aan deze basisregel voldoet.

22.10.8 Geheugenbeheer bij accessor methoden

Zoals in sectie 22.3.2 aangegeven dienen er in de implementatie van accessor methoden ook **retain** en **release** boodschappen verstuurd te worden (voor de niet-primitieve attributen). Volgende code komt uit het **A.m** bestand in sectie 22.3.2.

```
-(void) setStr:(NSString *)inputString
{
    [inputString retain];
    [str release];
    str = inputString;
}

-(void) setDate:(NSDate *)input
{
    [input retain];
    [date release];
    date = input;
}

-(void) setX:(int)input
{
    x = input;
}
```

Deze **retain** en **release** boodschappen zijn noodzakelijk om de eventueel reeds bestaande waarde van het attribuut in kwestie vrij te geven, en eigenaar te worden van de nieuwe doorgegeven waarde. Indien de huidige waarde **nil** is, kan het zoals vermeld in Objective-C geen kwaad om een boodschap te versturen naar een **nil** object (in tegenstelling tot C++).

Het **x** attribuut is van een primitief type (**int** in dit geval) en hiervoor dienen dus geen **retain** en **release** boodschappen verstuurd te worden.

22.10.9 Gebruik van properties

Om te vermijden dat men steeds voor elke accessor methode de implementatie zoals in vorige sectie dient te voorzien, kan men gebruikmaken van properties in Objective-C.

Aan de hand van een property worden de accessor methode gedeclareerd (zowel get- als set-methoden) in een headerbestand. De code hieronder illustreert de declaratie van accessor methoden via properties (in bestand A.h):

```
#import <Foundation/Foundation.h>

@interface A : NSObject
{
    NSString *str;
    NSDate *date;
    int x;
}

//accessor methods
@property(nonatomic, retain) NSString *str;
@property(nonatomic, retain) NSDate *date;
@property(nonatomic) int x;

...
@end
```

Tussen de ronde haakjes (()) worden de attributen van de properties vastgelegd. Mogelijkheden zijn:

- **nonatomic** of **atomic**: heeft te maken met synchronisatie bij applicaties met meerdere parallelle threads en het gedrag van een gelijktijdige **get** en **set** op hetzelfde attribuut. Default waarde (dus als deze niet explicet gespecificeerd wordt) is **atomic**. De nonatomic versie is echter veel sneller (geen synchronizatie vereist) en wordt zeer dikwijls gekozen (zoals in het voorbeeld hierboven).
- **retain** of **assign**: geeft aan of de klasse eigenaar moet worden (door een **retain** boodschap te versturen) of er gewoon een assignatie gebeurt. Default waarde is **assign**.
- **readonly** of **readwrite**: geeft aan of er enkel een get-methode (bij **readonly**) of er ook een set-methode dient gegenereerd te worden (bij **readwrite**). Default waarde is **readwrite**.

In het implementatie bestand (A.m in dit voorbeeld) wordt dan de implementatie gegenereerd aan de hand van de **synthesize** operatie, zoals hieronder getoond:

```
@implementation A
// =====
// = Accesssor methods =
// =====

@synthesize str, date, x;
...
@end
```


Deel VI

Appendices

Bijlage A

Dynamic Memory Management in C

In this appendix, the concepts of dynamic memory management in C are detailed. Important rules for dealing with dynamic memory allocation are listed, together with an overview of scenarios to illustrate the different rules.

A.1 Allocation

When designing software programs, there are two options when using variables or data structures:

1. it is known in advance (i.e. at compile time) how much memory is exactly required for using the variables and data structures: in this case a *static allocation* of the memory can take place.
2. it is *not* known in advance (i.e. at compile time) how much memory is exactly required for using the variables and data structures, only at execution time (i.e. at run time) it can be determined how much memory is exactly needed: in this case *dynamic allocation* of the memory has to take place.

In a single program, a combination of static and dynamic allocation can occur: some variables and data structures use static allocation, whereas *other* variables and data structures use dynamic allocation of the required memory.

For dynamic allocation of memory, the `malloc`, `calloc` and `realloc` functions should be used in C (available when including the `<stdlib.h>` header file).

It is the responsibility of the programmers in C to explicitly deallocate dynamically allocated memory when it is no longer needed. Programmers need to invoke the `free` function in C (also available when including the `<stdlib.h>` header file).

An example of static allocation in C:

```
int main(){  
    double d[10];  
    int i;
```

```

/* read the 10 values from file or via keyboard input */
for(i=0;i<10;i++){
    printf("d[%d]=%f\n",i,d[i]); /* print d[i] */
}
return 0;
}

```

The same example but with dynamic allocation:

```

int main(){
    double* d;
    int i;
    int nr_doubles;
    /* read the number of values from file, via keyboard input or
       as a command line argument via argv and store this value in
       nr_doubles */
    d=(double*)malloc(nr_doubles*sizeof(double));
    /* read the values either (1) from file, (2) via keyboard input
       or (3) as command line arguments via argv and store them in the
       d array*/
    for(i=0;i< nr_doubles;i++){
        printf("d[%d]=%f\n",i,d[i]); /* print d[i] */
    }
    free(d);
    return 0;
}

```

A.2 Memory structure

When a program is loaded into memory, it is organized into the following three *separate* areas of memory (each area is referred to as a segment):

1. the **text or code segment**: where the compiled code of the program is located. This is the machine language representation of the program steps to be executed, including all functions making up the program, comprising both the user defined functions and library functions.
2. the **stack segment**: where memory is allocated for *local* variables within functions. Parameters passed as *arguments* to functions are also copied to the stack memory, in that function's stack frame. When a function returns, the return value is copied to the calling context (= the stack frame of the calling function) and the memory for the local variables and arguments is deallocated (the called and now finished function's stack frame is deleted). The same stack memory can thus be reused when another function is called.
3. the **heap segment**: where memory can be allocated to be available until it is explicitly deallocated (usually it is made available for the duration of the program). Global variables (storage class **external**), and static variables (storage class **static**)

are allocated in this memory. Dynamically allocated memory (through the `malloc`, `calloc` and `realloc` functions in C) is also located on the heap. Once allocated it stays reserved until explicitly deallocated by the programmer (through invocation of the `free` function in C) or until the program finishes (but then we would have a *memory leak*).

More specific details about memory layout and structure are part of the course Computer Architecture and are not specifically required for this course.

A.3 Important rules

The following 7 important rules can be distinguished when dealing with dynamic memory management in C:

- I. Deallocation of memory through the `free` function in C should only take place on pointers to dynamically allocated memory (in the heap segment). The pointer to the dynamically allocated memory is obtained as a return value of the `malloc`, `calloc` or `realloc` functions in C. Note that a copy of this pointer value can of course also be used for deallocation of the memory.
- II. Allocated memory in the heap segment stays allocated until (i) a `free` (in C) is executed on the pointer to the allocated memory or (ii) the program finishes. It is considered a good programming practice to deallocate all allocated memory on the heap before a program finishes. For every invocation of a `malloc`, `calloc` or `realloc` function in C, there should be a corresponding invocation of the `free` function. In case this is not done, a **memory leak** is created.
- III. Do not use pointers that do not point to allocated memory (either in stack or heap segment). A pointer that doesn't contain a valid address and is different from `NULL` is referred to as a **dangling pointer**. Attempts to use a dangling pointer or `NULL` pointer for data access will produce a runtime memory access error.
- IV. When a function is invoked, a copy of the arguments is put into the stack frame for that function. This copy is used during the execution of the function. Local variables in a function are not visible to outside code, unless they are passed as return value or assigned to one of the arguments. Assignment to a copy of an argument of course doesn't change the original value of an argument. In case the original value of an argument variable needs to be adapted in a function, a pointer to the variable needs to be passed as argument to the function that needs to change this variable's value.
- V. Making a copy (via the assignment operator `=`) of a `struct` variable, will copy all individual fields of the `struct`: when a `struct` variable contains a pointer as one of its fields, the same field of the copied `struct` variable will point to the same memory location. This is referred to as a **shallow copy** (Dutch: ondiepe kopie)

since two variables point to the same memory: invoking the `free` function to one of the fields, makes the other `struct` variable unusable, since it has been reduced to a dangling pointer! A better approach is a **deep copy** (Dutch: diepe kopie), which allocates new memory for the field where a copy can be stored: in this case the two variables are completely separate in memory and independent from each other.

- VI. The `realloc` function in C attempts to change the size of the allocated memory. A decrease of the size will always succeed; an increase might result in an error if not enough memory is available. In the latter case a `NULL` value will be passed as return value of the `realloc` function. When not enough (adjacent) memory is available for increasing the memory size, some compilers copy the memory to a place in memory where enough space is available for increasing the size to the requested value and return a pointer to this new location (in this case a `free` should not be invoked on the previous pointer value, since this is done automatically, but a `free` should of course be invoked on the updated pointer value when the variable corresponding to this pointer is no longer required).
- VII. The return values of the `malloc`, `calloc` and `realloc` function in C contain `NULL` when the operation is not successful: in case few memory is available on a device or huge amounts of memory are required, the return values should be carefully checked after each invocation.

A.4 Illustrative scenarios

SCENARIO 1: PASS ARGUMENTS TO FUNCTION, ARGUMENTS CAN NOT BE USED FOR RETURNING INFORMATION A function is used to create a string (array of chars) containing `n` chars, where `n` is an input parameter to the function.

```
void f (char* a, int n){
    char* t;
    int i;
    ...
    t=(char*)malloc((n+1)*sizeof(char));
    for(i=0;i<n;i++)
        t[i]=(char)('a'+i%26);
    t[n]='\0';
    a=t;
}
```

This function is invoked in the `main` function:

```
int main(){
    char* a;
    f(a, 10);
    printf("%s\n",a); /* error1! a is still uninitialized! */
    free(a); /* error2! a doesn't point to allocated heap memory*/
```

Appendix A: Dynamic Memory Management in C

```
    return 0;
}
```

The reason for the error1 is assignment to a copy of an argument (rule IV) and the use of the dangling pointer (rule III) in the `printf` function. The reason for the error2 is violation of rule I (deallocation of a variable which does not point to dynamically allocated memory). Since the dynamically allocated memory in the function is never deallocated rule II (introduction of memory leak) is also violated. A possible solution is to use the return value to return the result (scenario 2) or use an argument for properly returning a result (scenario 3).

SCENARIO 2: ALLOCATE MEMORY IN FUNCTION, RETURNED VIA RETURN VALUE

Consider the function `f` with a different signature (return type `char*` instead of `void`)

```
char* f (int n){
    char* t;
    int i;
    ...
    t=(char*)malloc((n+1)*sizeof(char));
    for(i=0;i<n;i++)
        t[i]=(char)('a'+i%26);
    t[n]='\0';
    return t;
}
```

When this function is invoked in the `main` function, the above problems of scenario 1 are now solved.

```
int main(){
    char* a;
    a=f(10);
    printf("%s\n",a); /* gives correct results*/
    free(a); /* important otherwise memory leak*/
    return 0;
}
```

It is important to invoke the `free` function on the variable `a` to avoid memory leaks (rule II). In a lot of cases the return type has to be an integer (i.e. in order to be able to pass an error code), which implies that the results have to be passed through the arguments (as illustrated in scenario 3).

SCENARIO 3: ALLOCATE MEMORY IN FUNCTION, ARGUMENTS CAN BE USED FOR RETURNING INFORMATION

In this case the signature of the function `f` is again slightly different (argument type `char**` instead of `char*`):

```
void f (char** a, int n){
    char* t;
    int i;
```

```

...
t=(char*)malloc((n+1)*sizeof(char));
for(i=0;i<n,i++)
    t[i]=(char)('a'+i%26);
t[n]='\0';
*a=t;
}

```

When invoking this function it works perfectly:

```

int main(){
    char* a;
    f(&a, 10);
    printf("%s\n",a); /* correct result! */
    free(a); /* OK */
    return 0;
}

```

It is important to note that in the function **f** no **free(t)** or **free(*a)** should be invoked, since this could result in an error when invoking the **printf** function (rule III, access to non allocated memory). If this were the case, the call to **free(a)** in the **main** function would then also result in an error, since memory can not be deallocated twice (violation of rule I).

SCENARIO 4: ALLOCATE MEMORY IN FUNCTION, NOT FREED NOR RETURNED
In case the function **f** doesn't return a pointer to allocated memory (either through return value or argument):

```

void f (char** a, int l){
    char* t;
    ...
    t=(char*)malloc(l*sizeof(char));
    ... /*no free nor assignment to *a here */
}

```

Invocation of this function results in a memory leak since there is no way of accessing (and consequently, deallocating) that memory after the function **f** has finished execution (rule II).

SCENARIO 5: ASSIGNMENT OF STRUCT VARIABLES

```

typedef struct{
    double* d;
    int l;
} array_struct;

main(){
    array_struct a1, a2;
    a1.l=10;
}

```

Appendix A: Dynamic Memory Management in C

```

a1.d=(double*)malloc(10*sizeof(double));
/* read the values either (1) from file, (2) via keyboard input
or (3) as command line arguments via argv and store them in the
a1.d array*/
a2=a1;
for(i=0;i< a1.l;i++){
    printf("a1.d[%d]=%f\n",i,a1.d[i]); /* print d[i] */
}
free(a1.d);
for(i=0;i< a2.l;i++){
    printf("a2.d[%d]=%f\n",i,a2.d[i]); /* error! */
}
}

```

Access to the pointer in struct variable **a2** after a **free** of **a1.d** results in a memory access error (dangling pointer, rule III) and more explicitly rule V (shallow versus deep copy) should be taken into account. The use of a deep copy is illustrated in scenario 6 (in a wrong way) and scenario 7 (in a correct way).

SCENARIO 6: ASSIGNMENT OF STRUCT VARIABLES THROUGH WRONG FUNCTION INVOCATION

```

typedef struct{
    double* d;
    int l;
} array_struct;

int deep_copy_array_struct(array_struct to, array_struct from){
    int i;
    to.l=from.l;
    to.d=(double*)malloc(from.l*sizeof(double));
    for(i=0;i<from.l;i++)
        to.d[i]=from.d[i];
    return 0;
}

main(){
    array_struct a1, a2;
    a1.l=10;
    a1.d=(double*)malloc(10*sizeof(double));
    /* read the values either (1) from file, (2) via keyboard input
    or (3) as command line arguments via argv and store them in the
    a1.d array*/
    deep_copy_array_struct(a2,a1);
    for(i=0;i< a1.l;i++){
        printf("a1.d[%d]=%f\n",i,a1.d[i]); /* print d[i] */
    }
    free(a1.d);
    for(i=0;i< a2.l;i++) {

```

```

        printf("a2.d[%d]=%f\n", i, a2.d[i]); /* error! */
    }
    free(a2.d); /* error! */
}

```

Access to the pointer in struct variable `a2` results in a memory access error since `a2` is never initialized (dangling pointer, rule III), caused by wrong argument type (rule IV). Note that invoking `free(a2.d)` in the `main` function will also result in an error, since memory can not be deallocated twice (violation of rule I).

SCENARIO 7: ASSIGNMENT OF STRUCT VARIABLES THROUGH CORRECT FUNCTION INVOCATION When using a different signature for the function `deep_copy_array_struct` (`array_struct*` instead of `array_struct`), it works perfectly.

```

typedef struct{
    double* d;
    int l;
} array_struct;

int deep_copy_array_struct(array_struct* to, array_struct from){
    int i;
    to->l=from.l;
    to->d=(double*)malloc(from.l*sizeof(double));
    for(i=0;i<from.l;i++)
        (to->d)[i]=from.d[i];
    return 0;
}

main(){
    array_struct a1, a2;
    a1.l=10;
    a1.d=(double*)malloc(10*sizeof(double));
    /* read the values either (1) from file, (2) via keyboard input
    or (3) as command line arguments via argv and store them in the
    a1.d array*/
    deep_copy_array_struct(&a2,a1);
    for(i=0;i< a1.l;i++){
        printf("a1.d[%d]=%f\n", i, a1.d[i]); /* print d[i] */
    }
    free(a1.d);
    for(i=0;i< a2.l;i++){
        printf("a2.d[%d]=%f\n", i, a2.d[i]); /* correct result! */
    }
    free(a2.d); /* important, otherwise memory leak!*/
}

```

Please note that in all illustrative scenarios, the return values of the `malloc` functions are never checked to be different from `NULL` (cfr. rule VII) since it is assumed that enough memory is available for these examples.

Bijlage B

Overzicht Eigenschappen C++11

B.1 Inleiding

C++11 is de meest recente versie van de C++ programmeertaal en werd op 12 augustus 2011 door de International Organization for Standardization (ISO) erkend als standaard. C++11 voegt vele zaken toe aan de kern van C++ (met focus op performantie- en bruikbaarheidsverbeteringen) en breidt de standaard bibliotheek uit. Experimentele ondersteuning voor C++11 is voorzien voor GCC 4.5 en later, Microsoft Visual C++ 11 (erg beperkte implementatie en op sommige punten afwijkend van de standaard) en versie 11 van de Intel compiler. In deze appendix zullen we de voornaamste eigenschappen van de C++11 standaard toelichten, in vergelijking met de C++ versie, die eerder in deze cursus uitgebreid aan bod kwam.

B.2 Automatische type-afleiding en decltype

In voorgaande versies van de C++ standaard dien je altijd het type van een object expliciet te specifiëren wanneer je het object declareert. Vaak wordt een objectdeclaratie echter vergezeld van een initialisatie. C++11 laat toe om objecten te declareren zonder het type te specifiëren, aan de hand van het `auto` sleutelwoord:

```
auto a=0; //a heeft type int want 0 is int
auto b='a'; //b heeft type char
auto c=0.5; //c heeft type double
auto d=1440000000000LL;//long long
```

Dit automatisch afleiden van type is vooral nuttig wanneer het type van het object zeer complex is (het verhoogt de leesbaarheid van de code) of wanneer dit type automatisch gegenereerd wordt (bijv. bij gebruik van templates). Vroeger moest volgende typedeclaratie uitgevoerd worden bij het initialiseren van een const iterator (const iterators laten iteratie toe over elementen van een STL container, maar geen manipulatie) voor een STL vector:

```
void func(const vector<int> &v){
    vector<int>::const_iterator ci=v.begin();
}
```

In C++11 kan diezelfde iterator als volgt gedeclareerd worden:

```
auto ci=v.begin();
```

Het `auto` sleutelwoord is niet nieuw, maar C++11 heeft de betekenis ervan veranderd. `auto` slaat niet langer op een object met automatisch opslagtype, maar duidt op een object waarvan het objecttype afgeleid kan worden op basis van de initialisatie-uitdrukking.

C++11 biedt ook een mechanisme aan voor het ophalen van een objecttype of expressietype. De nieuwe operator `decltype` geeft het type van een expressie terug. Dit is handig in combinatie met de declaratie van een nieuw type aan de hand van `typedef`, zoals hieronder geillustreerd:

```
const vector<int> v;
typedef decltype(v.begin()) CIT; //definitie nieuw type const_iterator
CIT another_const_iterator;
```

B.3 Uniforme initialisatiesyntax

Traditioneel laat C++ vier verschillende initialisatienotaties toe:

1. Geparenthiseerde initialisatie ziet er als volgt uit:

```
std::string s("hello");
```

2. Men kan ook de '=' notatie gebruiken in bepaalde gevallen:

```
std::string s="hello";
```

3. Voor POD (Plain Old Data) aggregaten, kan men haakjes gebruiken:

```
int arr[4]={0,1,2,3};
```

4. Finaal kunnen constructoren members als volgt initialiseren:

```
class S {
    int x;
public:
    S(): x(0) {}
};
```

Al deze verschillende initialisatienotaties kunnen zorgen voor verwarring. In voorgaande versies van de C++ standaard kan men bovendien geen Plain Old Data array's initialiseren die gealloceerd werden met behulp van de `new[]` operator. C++11 verbetert dit door het invoeren van een uniforme haakjes-notatie:

```

class C {
    int a;
    int b;
public:
    C(int i, int j);
};

C c {0,0}; //enkel in C++11, equivalent aan: C c(0,0);
int* a = new int[3] { 1, 2, 3 }; // enkel in C++11

class X {
    int a[4];
public:
    X() : a{1,2,3,4} {} //enkel in C++11, member array initialisatie
};

```

Ook bij het gebruik van containers hoef je de initialisatie niet langer met een lange lijst van `push_back()` oproepen uit te voeren. In C++11 kunnen containers als volgt geïnitialiseerd worden:

```

vector<string> v={"eerste","tweede","derde"};
map<string,string> adressen = {{"Bart","bart@gmail.com"},
                                {"Jan","jan@facebook.com"}};

```

C++11 laat ook in-klasse initialisatie van de data members toe:

```

class C {
    int a = 7; //enkel in C++11 mogelijk
public:
    C();
};

```

B.4 Deleted en Defaulted functies

Methoden in de volgende vorm:

```

class A {
A()=default; //enkel in C++11
virtual ~A()=default; //enkel in C++11
};

```

worden 'defaulted' functies genoemd. Het `=default;` gedeelte instrueert de compiler om de default implementatie voor deze functie te genereren. Defaulted functies hebben twee voordelen: ze zijn efficiënter dan manuele implementaties, en ze zorgen dat de programmeur deze functies niet manueel moet definiëren. Bovendien is het onmiddellijk duidelijk bij inspectie van de code dat de implementatie van de functie niet door de programmeur wordt voorzien, maar dat er gerekend wordt op de compiler om een default implementatie te genereren.

Het tegenovergestelde van een 'defaulted' functie is een 'deleted' functie:

```
int func()=delete;
```

Deleted functies zijn nuttig om bijvoorbeeld te verhinderen dat objecten gekopieerd worden. In C++ wordt namelijk automatisch een copy-constructor en assignment-operator voor klassen gedefinieerd. Om dit te vermijden kan je die twee speciale member functies met `=delete;` declareren:

```
class NoCopy {
    NoCopy & operator =(const NoCopy &) =delete;
    NoCopy (const NoCopy &) =delete;
};

NoCopy a;
NoCopy b(a); //compilatie-fout, copy constructor werd deleted.
```

B.5 nullptr

C++11 beschikt over een sleutelwoord dat een null-pointer constante voorstelt: `nullptr` vervangt de `NULL` macro en 0 die gebruikt werden als null-pointer substituten. `nullptr` is bovendien sterk getypeerd, hetgeen dubbelzinnigheden vermindert, zoals hieronder geïllustreerd:

```
void func(int); //functie #1
void func(char*); //functie #2
func(0); //welke functie werd opgeroepen? onduidelijk

//volgende regel enkel geldig in C++11
func(nullptr) //roep functie #2 op, sterk getypeerde nullptr
```

B.6 Delegerende constructoren

In C++11 mag een constructor een andere constructor van dezelfde klasse oproepen:

```
class M {
    int x, y;
    char *p;
public:
    M(int v):x(v),y(0),p(new char [MAX]){} // constructor #1
    M():M(0){cout<<"delegating constructor"<<endl;} // constructor #2
};
```

Constructor 2, de delegerende constructor, invokeert constructor 1.

B.7 Rvalue Referenties

In wat volgt zullen we summier een geavanceerd nieuw element van C++11 behandelen, dat ingezet kan worden om een zo geoptimaliseerd mogelijke performantie te bekomen:

Appendix B: Overzicht Eigenschappen C++11

rvalue referenties. Referentietypes in voorgaande versies van de C++ standaard kunnen enkel binden met **lvalues**. C++11 introduceert een nieuwe categorie van referentietypes genaamd **rvalue**-referenties. **rvalue**-referenties kunnen met **rvalues** (tijdelijke objecten en literals) binden.

De voornaamste reden om **rvalue**-referenties toe te voegen is om zogenaamde verplaats (Eng.:move)-semantiek te ondersteunen. In tegenstelling tot traditioneel kopiëren, betekent verplaatsen dat een doel-object de resources van het bron-object overneemt, en zo de bron in een 'lege' toestand achterlaat. In bepaalde gevallen is het maken van een effectieve kopie van een object zowel duur als onnoodig, en kan beter een verplaats-operatie gebruikt worden. Om een voorbeeld te geven van de performantiewinst bij het gebruik van deze techniek, beschouwen we het verwisselen van strings. Een naïeve implementatie ziet er als volgt uit:

```
void naiveswap(string &a, string & b){  
    string temp=a;  
    a=b;  
    b=temp;  
}
```

Dit is inherent een kostelijke operatie. Het kopiëren van een string bevat de allocatie van geheugen (**string temp = a**) en het kopiëren van karakters één voor één van bron naar bestemming. Indien strings 'verplaatst' worden, worden in feite twee datamembers omgewisseld, zonder geheugen te hoeven alloceren, zonder karakters één voor één te hoeven kopiëren en zonder geheugen terug vrij te moeten geven. Volgend voorbeeld (in pseudocode om niet in de interne complexe code van string te hoeven werken) toont aan wat deze nieuwe techniek achterliggend doet:

```
void moveswapstr(string& a, string & b) {  
    size_t sz=a.size();  
    const char *p= a.data(); //pointer naar karakter data van a  
    //verplaats resources (members) van b naar a  
    a.setsize(b.size());  
    a.setdata(b.data());  
    //b wordt a  
    b.setsize(sz);  
    b.setdata(p); //karakter data van b wijst nu naar karakter data van a  
}
```

Indien je een klasse implementeert waarbij 'verplaatsen' gesupporteerd moet worden, kan een **move** constructor en een **move** assignatie-operator als volgt gedeclareerd worden (dubbele & notatie):

```
class Movable {  
    Movable (Movable&&); //move constructor  
    Movable&& operator=(Movable&&); //move assignatie-operator  
};
```

In het voorbeeld hieronder wordt een **move**-constructor geïmplementeerd voor een eenvoudige **ArrayWrapper** klasse:

```

class ArrayWrapper {
public:
//default constructor
ArrayWrapper ():_p_vals(new int[64]),_size(64){}
    ArrayWrapper (int n):_p_vals(new int[n]),_size(n){}

// copy constructor
ArrayWrapper (const ArrayWrapper& other):
    _p_vals(new int[other._size]),
    _size(other._size){
        for (int i = 0; i < _size; ++i){
            _p_vals[i] = other._p_vals[i];
        }
    }

~ArrayWrapper () {
    delete [] _p_vals;
}

private:
    int *_p_vals; //array waarden
    int _size; //grootte van de array
};

```

Let er op dat de copy constructor zowel geheugen moet aanmaken als elke waarde van de array moet kopiëren. Laat ons nu een move constructor toevoegen:

```

class ArrayWrapper {
public:
// default constructor
ArrayWrapper ():_p_vals(new int[64]),_size(64){}
    ArrayWrapper (int n):_p_vals(new int[n]),_size(n){}

// move constructor
ArrayWrapper (ArrayWrapper&& other):
    _p_vals(other._p_vals),
    _size(other._size){
        other._p_vals = NULL;
    }

// copy constructor
ArrayWrapper (const ArrayWrapper& other):
    _p_vals(new int[other._size]),
    _size(other._size){
        for ( int i = 0; i < _size; ++i ){
            _p_vals[ i ] = other._p_vals[ i ];
        }
    }
}

```

```

~ArrayWrapper () {
    delete [] _p_vals;
}

private:
    int *_p_vals;
    int _size;
};

```

De `move`-constructor is dus eenvoudiger dan de `copy`-constructor. Belangrijkste zaken om op te merken zijn: de parameter is een niet-constante `rvalue` referentie (i.e. een tijdelijk object) en `other._p_vals` wordt gelijkgesteld aan `NULL`. De parameter is namelijk een tijdelijk object, dat, net zoals alle overige C++ objecten, vernietigd zal worden eens het `out of scope` gaat. Het zal dus `_p_vals` proberen vrijgeven, dezelfde `_p_vals` die we net aan een nieuw object toegekend hebben. Vandaar dat we deze pointer naar `NULL` laten wijzen, om te vermijden dat dit geheugen vrijgegeven wordt.

De C++11 standaard bibliotheek maakt veelvuldig gebruik van verplaats (Eng.:move)-semantiek. Vele algoritmes en containers zijn nu `move`-geoptimaliseerd, hetgeen ervoor zorgt dat C++11 nog betere performantie vertoont dan voorgaande versies.

B.8 Nieuwe Smart Pointer klassen

Smart pointers kunnen gezien worden als 'pointer wrapper' klassen die naast de pointer zelf ook additionele eigenschappen aanbieden zoals het automatisch vrijgeven van geheugen indien de pointer niet meer gebruikt wordt of controle op het overschrijden van geheugengrenzen. In een voorgaande versie van de C++ standaard (C++98) werd slechts één smart pointer klasse gedefinieerd, `auto_ptr`, die nu deprecated is. C++11 includeert een nieuwe smart pointer klasse: `shared_ptr` en de recent toegevoegde `unique_ptr`. Beiden zijn compatibel met de overige standard library componenten, dus je mag de smart pointers in standaard containers plaatsen en deze manipuleren met standaard algoritmes.

`unique_ptr` is een smart pointer die ten alle tijde slechts 1 'eigenaar' van de pointer afdwingt. Dit type smart pointer kan dus niet gekopieerd worden, maar ondersteunt wel het wijzigen van eigenaar via `move`-semantiek, zoals hieronder geïllustreerd:

```

//nieuwe unique_ptr die naar rij v. 5 integers wijst
std::unique_ptr<int> p1(new int(5));
//compilatiefout, unique_ptr kan niet gekopieerd worden.
std::unique_ptr<int> p2 = p1;
//transfereert eigenaar via move-semantiek
std::unique_ptr<int> p3 = std::move(p1);
p3.reset(); //geeft geheugen vrij
p1.reset(); //doet niets

```

In tegenstelling tot `unique_ptr` kan een `shared_ptr`, zoals de naam doet vermoeden, wel door meerdere eigenaren gedeeld worden (door middel van het zogenaamde `reference counting` mechanisme). `shared_ptr` zal het geheugen waarnaar gewezen wordt slechts vrijgeven als alle instanties van die `shared_ptr` vernietigd zijn.

```

std::shared_ptr<int> p1(new int(5));
std::shared_ptr<int> p2 = p1; //beiden zijn nu 'eigenaar' van dit geheugen
//geheugen nog niet vrijgegeven, p2 wijst hier nog naar.
p1.reset();
//geheugen wordt vrijgegeven, er zijn geen eigenaars meer
//die er nog gebruik van maken.
p2.reset();

```

B.9 Anonieme functies: Lambda expressies

Een Lambda-expressie laat toe om functies lokaal te definiëren, namelijk op de plaats van de functie-oproep. Dit mechanisme stelt ons in staat om sneller te werken dan bij het gebruik van benoemde functie-objecten of functiepointers. Een Lambda-expressie wordt als volgt gedefinieerd:

```
[capture] (parameters)->return-type {body}
```

waarbij **capture** de opsomming bevat van variabelen die in de Lambda expressie kunnen aangepast worden, **parameters** de argumenten van de Lambda expressie aangeeft, **return-type** ingevuld wordt met het return-type (tenzij het automatisch kan afgeleid worden als er bijv. geen of slechts één **return** uitdrukking tussen de accolades voorkomt), en **body** de eigenlijke implementatie van de Lambda expressie bevat.

Stel dat we een zoekfunctie willen opstellen voor een klasse **Adresboek** die e-mailadressen van gebruikers bijhoudt. Dan kunnen we een zoekfunctie implementeren die een string als parameter neemt en alle adressen waar deze string in voorkomt retourneert. In bepaalde gevallen zal dergelijke zoekfunctie voldoende zijn. De code hieronder illustreert dit:

```

#include <string>
#include <vector>

class Adresboek {
public:
    // gebruik van template laat toe functies, functie-pointers
    // en lambda-expressies te gebruiken
    template<typename Func>
    std::vector<std::string> zoekAdres (Func func){
        std::vector<std::string> resultaat;
        for(auto itr = _adressen.begin(), end = _adressen.end(); itr!=end; ++itr ) {
            // functie oproepen die als argument aan zoekAdres meegegeven werd
            if (func(*itr)) {
                resultaat.push_back( *itr );
            }
        }
        return resultaat;
    }
private:

```

```
    std::vector<std::string> _adressen;
};
```

Wanneer de functie die als parameter aan `zoekAdres` meegegeven wordt `true` teruggeeft zal het adres in de vector `resultaat` toegevoegd worden. In traditionele C++ moet men voor elke mogelijk zoekfunctie op een andere locatie in de code een nieuwe functie definiëren. Om dit te vermijden werden Lambda expressies gecreëerd. Hieronder wordt een basisvoorbeeld voor een Lambda expressie gegeven:

```
#include <iostream>
using namespace std;

int main() {
    auto func = [] () { cout << "Hello world"; };
    func();
}
```

De vierkante haken `[]` geven aan waar de Lambda-expressie start. De hierna volgende ronde haken `()` geven de parameterlijst aan (die hier leeg is). Het lichaam van de Lambda-expressie wordt vervolgens gedefinieerd tussen `{}`. De return waarde van deze Lambda expressie kan automatisch afgeleid worden, en hoeft dus niet gespecificeerd te worden.

In het bovenstaande adresboekvoorbeeld kunnen Lambda-expressies als volgt gebruikt worden om alle e-mails terug te vinden die afkomstig zijn uit het `.org` domein:

```
Adresboek globaal_adresboek;
vector<string> zoekAdressenUitDomein () {
    return globaal_adresboek.zoekAdres([](const string& addr){
        return addr.find(".org")!=string::npos;})
    );
}
```

Deze Lambda-expressie starten we opnieuw met `[]` en deze keer hebben we een argument namelijk het adres waarbij we wensen te controleren als het e-mailadres afkomstig is uit het domein `.org`. Bij elke oproep van `zoekAdres` zal de Lambda-expressie opengroepen worden met het e-mailadres als argument, en zal de gegeven Lambda-expressie controleren of dit e-mailadres aan de voorwaarden voldoet.

B.10 C++11 Standard Library

C++ heeft grote veranderingen ondergaan in 2003 onder de vorm van de Library Technical Report 1 (TR1). TR1 introduceerde enkele nieuwe container klassen (`unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap`) en verschillende nieuwe bibliotheken voor reguliere expressies, tuples, functie-object wrappers en meer. Met de goedkeuring van C++11 is TR1 nu officieel geïncorporeerd in de C++ standaard. Verder werden ook nieuwe bibliotheken en algoritmes toegevoegd aan de C++11 standaard bibliotheek.

B.10.1 Nieuwe algoritmes

De C++11 Standard Library definieert nieuwe algoritmes die de verzamelingsleer-operaties `all_of()`, `any_of()` en `none_of()` implementeren. Het volgende voorbeeld past het predicaat `ispositive()` toe op het bereik `[first, first+n]` (`first` is hierbij een iterator) en gebruikt `all_of()`, `any_of()` en `none_of()` om de eigenschappen van de elementen in dit bereik te inspecteren:

```
#include <algorithm>
//enkel in C++11
//zijn alle elementen positief?
all_of(first, first+n, ispositive());
//is er minstens n element positief?
any_of(first, first+n, ispositive());
//is geen enkel element positief?
none_of(first, first+n, ispositive());
```

Een nieuwe categorie van `copy_n` algoritmes is ook beschikbaar. Gebruik makend van `copy_n()`, wordt het kopiëren van bvb. een array van 5 elementen naar een andere array, zeer gemakkelijk.

```
#include <algorithm>
int source[5]={0,12,34,50,80};
int target[5];
//kopieer 5 elementen van source naar target
copy_n(source,5,target);
```

Het algoritme `iota()` creëert een bereik van sequentieel vermeerderende waarden, alsof een initiële waarde aan `*first` toegekend wordt, om dan die waarde te incrementeren gebruik makend van de `++-prefix operator`. In het volgende voorbeeld zal `iota()` de opeenvolgende waarden `10,11,12,13,14` aan array `a` toekennen, en `'a','b','c'`, aan char array `c`.

```
include <numeric>
int a[5]={0};
char c[3]={0};
iota(a, a+5, 10); //wijzigt a naar {10,11,12,13,14}
iota(c, c+3, 'a'); //wijzigt c naar {'a','b','c'}
```

B.11 Threading bibliotheek

Eén van de belangrijkste nieuwe features in de C++11 standaard is ondersteuning voor multi-threading. Voorheen werd multithreading ondersteund als een extensie van de C++ standaard, wat met zich meebracht dat ondersteuning ervoor verschilde tussen compilers en platformen.

Met C++11 zal elke compiler moeten voldoen aan hetzelfde geheugenmodel en zullen ze dezelfde faciliteiten voor multi-threading moeten voorzien (ze mogen wel extensies

voorzien). Dit betekent dat je multi-threaded code tussen compilers en platformen zal kunnen porteren met heel wat minder effort. Het betekent ook dat er minder kennis van verschillende platformspecifieke APIs en syntaxes zal nodig zijn wanneer men op meerdere platformen wil kunnen uitrollen.

De kern van de nieuwe thread library is de `std::thread` klasse, die een uitvoerings-thread beheert.

B.11.1 Het lanceren van threads

Een nieuwe thread wordt gestart door het aanmaken van een instantie van `std::thread` met als argument een functie. Deze functie wordt dan gebruikt als begin-uitvoerpunt voor de nieuwe thread, en eens de functie retourneert is de thread afgelopen.

```
void do_work();
std::thread t(do_work);
```

Men is hierbij niet gelimiteerd tot het doorgeven van functies alleen. Zoals veel algoritmes in de standaard C++ bibliotheek, zal `std::thread` een object van een type dat de functie-oproep operator (`operator()`) implementeert accepteren:

```
class do_work {
public:
    void operator()();
};

do_work dw;
std::thread t(dw);
```

Het is belangrijk op te merken dat dit het aangeleverde object in de thread kopiëert. Indien je dit object dan ook wenst te manipuleren (waarbij je er zeker van moet zijn dat dit object niet vernietigd wordt voordat de thread finaliseert), kan je dit doen door het te wrappen in `std::ref` om zodoende een referentie naar dit object te bekomen. `std::ref` geeft in feite een referentie-wrapper van zijn argument terug: een object dat over referentie-semantiek beschikt (het oproepen van een member functie op een referentie-wrapper roept die member functie op het onderliggende object op).

```
do_work dw;
std::thread t(std::ref(dw));
```

De meeste bestaande thread-creatie APIs laten toe om één enkele parameter aan een nieuw gecreëerde thread toe te voegen, typisch een `long` of een `void*`. `std::thread` laat ook argumenten toe, maar laat een willekeurig aantal toe en van om het even welk (kopieerbaar) type. Volgende voorbeeld toont dat je objecten van om het even welk kopieerbaar type als argument aan de thread-functie mee kan geven:

```
void do_more_work(int i,std::string s,std::vector<double> v);
std::thread t(do_more_work,42,"hello",std::vector<double>(23,3.141));
```

Zoals bij functie-objecten zelf worden de argumenten in de thread gekopieerd voordat de functie uitgevoerd wordt. Wens je dus door middel van referentie door te geven dan dien je het argument met `std::ref` wrappen.

```
void foo(std::string&);  
std::string s;  
std::thread t(foo, std::ref(s));
```

Tot zover een korte introductie tot het lanceren van threads. Welke faciliteiten zijn er nu om threads te laten eindigen? De C++11 standaard noemt dat het `joinen` met de thread, en dit gebeurt via de `join()` member functie:

```
void do_work();  
std::thread t(do_work);  
t.join();
```

Het hoofdprogramma wacht bij de `t.join()` oproep tot alle werk in thread `t` uitgevoerd is, om dan verder te gaan. Indien `t` reeds uitgevoerd is op het moment dat `t.join` opgeroepen wordt, dan gaat het programma gewoon verder.

Als je op geen enkel moment van plan bent om te wachten tot een aangemaakte thread afgelopen is (i.e. te `joinen`) en je bent enkel verantwoordelijk voor het succesvol lanceren van die thread, kan je het thread-object `detachen`. Het oproepen van `detach()` geeft aan dat, eens je thread aangemaakt is, er geen verdere informatie benodigd is (bijv. wanneer en met welke status deze afloopt):

```
void do_work();  
std::thread t(do_work);  
t.detach();
```

B.11.2 Beschermen van data

In de C++11 thread library is de basisfaciliteit om gedeelde data te beschermen de `mutex` (naamgeving afgeleid van mutual exclusion). Mutexen worden gebruikt om data-inconsistentie te vermijden doordat meerdere threads gelijktijdig operaties zouden uitvoeren op hetzelfde deel van het geheugen of om race condities tussen verschillende threads te vermijden. Mutexen maken dus geserialiseerd gebruik van gedeelde resources als geheugen op een veilige manier mogelijk. De basis-mutex in C++11 is `std::mutex`. Er is ook nood aan een synchronisatie-mechanisme dat de toegang tot die mutex regelt. Dit synchronisatiemechanisme is een zogenaamde 'lock': indien een thread data, afgeschermd door een mutex, wenst te gebruiken, dan zal die thread eerst een lock proberen verkrijgen op die mutex (indien dit niet mogelijk is betekent het namelijk dat een andere thread momenteel gebruik maakt van de data afgeschermd door die mutex) om daarna de gewenste bewerkingen op die data uit te voeren en de lock op de mutex vrij te geven. In C++ is het in de meeste gevallen best om met de `lock` class templates `std::unique_lock<>` en `std::lock_guard<>` mutexen te locken. Deze klassen locken

mutexen in de constructor en geven deze vrij in de destructor. Als je mutexen dan gebruikt als lokale variabelen worden deze automatisch unlocked wanneer die variabelen out-of-scope gaan. Dit wordt hieronder geïllustreerd:

```
std::mutex m;
my_class data;
void foo() {
    //we nemen lock op mutex m
    std::lock_guard<std::mutex> lk(m);
    process(data);
} // lock gaat out-of-scope -> mutex m unlocked
```

`std::lock_guard` werd principieel eenvoudig opgesteld en kan enkel gebruikt worden om een standaard lock te nemen op een mutex, zoals in het bovenstaande voorbeeld getoond werd. `std::unique_lock` is een meer geavanceerd locking mechanisme, dat toelaat om uitgesteld te locken, te proberen locken (met eventuele timeout) en te unlocken voordat het object vernietigd wordt (voor gedetailleerde uitleg wordt verwezen naar de C++11 standaard).

B.11.3 Ondersteuning voor thread-events

Indien er data gedeeld wordt tussen threads, gebeurt het vaak dat één thread moet wachten tot een andere thread een bepaalde actie uitgevoerd heeft, en dit zonder CPU-tijd te consumeren. Als een thread eenvoudigweg wacht op zijn beurt om toegang te krijgen tot gedeelde data, dan kan een mutex lock voldoende zijn.

De eenvoudigste manier om dit te verwezenlijken is om een thread voor een korte tijdspanne te laten slapen. Eens de thread terug wakker wordt kan gecontroleerd worden of de gewenste actie reeds ondernomen werd. De mutex die gebruikt wordt om data te beschermen, moet dus op een gegeven moment unlocked worden terwijl de wachtende thread slaapt.

```
std::mutex m;
bool data_ready;
void process_data();
void foo() {
    std::unique_lock<std::mutex> lk(m);
    while(!data_ready){
        lk.unlock(); //unlocken van mutex tijdens slapen van deze thread
        std::this_thread::sleep_for(std::chrono::milliseconds(10)); //slaap 10 ms
        lk.lock(); //locken van mutex bij het ontwaken van deze thread
    }
    process_data();
}
```

Deze manier is eenvoudig, maar zeker niet ideaal omwille van twee redenen. Eerst en vooral, een thread zal gemiddeld vijf milliseconden wachten om wakker te worden eens `data_ready` klaar is. Dit kan mogelijk te veel wachttijd introduceren. Hoewel dit op

zijn beurt verbeterd kan worden door de wachttijd te reduceren, geeft het een tweede probleem aan: de thread moet elke 10 ms wakker worden en de mutex vlag controleren, zelfs als niets gewijzigd is. Dit verbruikt CPU-tijd, verhoogt de contentie op de mutex, en vertraagt dus mogelijks ook de thread waarop we aan het wachten zijn.

Om dit te voorkomen kunnen conditievariabelen gebruikt worden. In plaats van een thread te laten slapen voor een vaste periode, kan je een thread laten slapen tot die wakker gemaakt wordt door een andere thread. Dit zorgt ervoor dat de wachttijd tussen notificatie en wakker worden van de thread zo kort mogelijk is (zo kort als het besturingssysteem toelaat), en verminderd bovendien de CPU-consumptie van de wachtende thread tot nul gedurende de ganse wachtperiode. Bovenstaand voorbeeld kan als volgt herschreven worden om een wachtconditie te introduceren:

```
std::mutex m;
//om conditie-wijziging te kunnen monitoren
std::condition_variable cond;
bool data_ready;
void process_data();
void foo() {
    std::unique_lock<std::mutex> lk(m);
    while(!data_ready){
        cond.wait(lk);
    }
    process_data();
}
```

Let er op dat bovenstaande code het lock object `lk` als een parameter aan `wait()` toevoegt. Deze implementatie unlockt de mutex bij toegang tot `wait()`, en lockt deze opnieuw bij exit. Dit zorgt ervoor dat de beveiligde data gemodificeerd kan worden door andere threads terwijl deze thread aan het wachten is. De code die de `data_ready` vlag behandelt, ziet er dan uit als volgt:

```
void set_data_ready(){
    std::lock_guard<std::mutex> lk(m);
    data_ready=true;
    cond.notify_one(); //conditiewijziging aangeven
}
```

B.12 Conclusie

Aan de C++ taal wordt nog dagelijks gesleuteld om de functionaliteit, het gebruiksgemak en de performantie te verhogen. In deze appendix hebben we enkele van de voornaamste wijzigingen van de C++11 standaard ten opzicht van voorgaande versies van de C++ standaard toegelicht. Het dient opgemerkt te worden dat veel van deze wijzigingen slechts preliminair (en met de nodige bugs) geïmplementeerd zijn in de huidige C++ compilers. Andere wijzigingen (zoals bijvoorbeeld extern templates, variadic templates, polymorfe wrappers voor functie-objecten) vallen buiten het bereik van deze cursus.

Bijlage C

C Traps and Pitfalls

Dit document geeft een overzicht van veelgemaakte fouten in C en bijhorende uitleg. De Nederlandse vertaling van de titel luidt: "C Vallen en Valkuilen".

Volgende secties van dit document zijn interessant om grondig door te nemen:

- sectie 1
- sectie 2.2 t.e.m. 2.6
- sectie 4.1 t.e.m. 4.3
- sectie 4.5 t.e.m. 4.7
- sectie 6

Andere secties zijn compiler-afhankelijk en dienen niet beschouwd te worden als onderdeel van de leerstof van deze cursus.

C Traps and Pitfalls*

Andrew Koenig

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The C language is like a carving knife: simple, sharp, and extremely useful in skilled hands. Like any sharp tool, C can injure people who don't know how to handle it. This paper shows some of the ways C can injure the unwary, and how to avoid injury.

0. Introduction

The C language and its typical implementations are designed to be used easily by experts. The language is terse and expressive. There are few restrictions to keep the user from blundering. A user who has blundered is often rewarded by an effect that is not obviously related to the cause.

In this paper, we will look at some of these unexpected rewards. Because they are unexpected, it may well be impossible to classify them completely. Nevertheless, we have made a rough effort to do so by looking at what has to happen in order to run a C program. We assume the reader has at least a passing acquaintance with the C language.

Section 1 looks at problems that occur while the program is being broken into tokens. Section 2 follows the program as the compiler groups its tokens into declarations, expressions, and statements. Section 3 recognizes that a C program is often made out of several parts that are compiled separately and bound together. Section 4 deals with misconceptions of meaning: things that happen while the program is actually running. Section 5 examines the relationship between our programs and the library routines they use. In section 6 we note that the program we write is not really the program we run; the preprocessor has gotten at it first. Finally, section 7 discusses portability problems: reasons a program might run on one implementation and not another.

1. Lexical Pitfalls

The first part of a compiler is usually called a *lexical analyzer*. This looks at the sequence of characters that make up the program and breaks them up into *tokens*. A token is a sequence of one or more characters that have a (relatively) uniform meaning in the language being compiled. In C, for instance, the token `->` has a meaning that is quite distinct from that of either of the characters that make it up, and that is independent of the context in which the `->` appears.

For another example, consider the statement:

```
if (x > big) big = x;
```

Each non-blank character in this statement is a separate token, except for the keyword `if` and the two instances of the identifier `big`.

In fact, C programs are broken into tokens twice. First the preprocessor reads the program. It must tokenize the program so that it can find the identifiers, some of which may represent macros. It must then replace each macro invocation by the result of evaluating that macro. Finally, the result of the macro replacement is reassembled into a character stream which is given to the compiler proper. The compiler then breaks the stream into tokens a second time.

* This paper, greatly expanded, is the basis for the book *C Traps and Pitfalls* (Addison-Wesley, 1989, ISBN 0-201-17928-8); interested readers may wish to refer there as well.

In this section, we will explore some common misunderstandings about the meanings of tokens and the relationship between tokens and the characters that make them up. We will talk about the preprocessor later.

1.1. = is not ==

Programming languages derived from Algol, such as Pascal and Ada, use := for assignment and = for comparison. C, on the other hand, uses = for assignment and == for comparison. This is because assignment is more frequent than comparison, so the more common meaning is given to the shorter symbol.

Moreover, C treats assignment as an operator, so that multiple assignments (such as a=b=c) can be written easily and assignments can be embedded in larger expressions.

This convenience causes a potential problem: one can inadvertently write an assignment where one intended a comparison. Thus, this statement, which looks like it is checking whether x is equal to y:

```
if (x = y)
    foo();
```

actually sets x to the value of y and then checks whether that value is nonzero. Or consider the following loop that is intended to skip blanks, tabs, and newlines in a file:

```
while (c == ' ' || c = '\t' || c == '\n')
    c = getc (f);
```

The programmer mistakenly used = instead of == in the comparison with '\t'. This “comparison” actually assigns '\t' to c and compares the (new) value of c to zero. Since '\t' is not zero, the “comparison” will always be true, so the loop will eat the entire file. What it does after that depends on whether the particular implementation allows a program to keep reading after it has reached end of file. If it does, the loop will run forever.

Some C compilers try to help the user by giving a warning message for conditions of the form $e1 = e2$. To avoid warning messages from such compilers, when you want to assign a value to a variable and then check whether the variable is zero, consider making the comparison explicit. In other words, instead of:

```
if (x = y)
    foo();
```

write:

```
if ((x = y) != 0)
    foo();
```

This will also help make your intentions plain.

1.2. & and | are not && or ||

It is easy to miss an inadvertent substitution of = for == because so many other languages use = for comparison. It is also easy to interchange & and &&, or | and ||, especially because the & and | operators in C are different from their counterparts in some other languages. We will look at these operators more closely in section 4.

1.3. Multi-character Tokens

Some C tokens, such as /, *, and =, are only one character long. Other C tokens, such as /* and ==, and identifiers, are several characters long. When the C compiler encounters a / followed by an *, it must be able to decide whether to treat these two characters as two separate tokens or as one single token. The C reference manual tells how to decide: “If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.” Thus, if a / is the first character of a token, and the / is immediately followed by a *, the two characters begin a comment, *regardless* of any other context.

The following statement looks like it sets `y` to the value of `x` divided by the value pointed to by `p`:

```
y = x/*p /* p points at the divisor */;
```

In fact, `/*` begins a comment, so the compiler will simply gobble up the program text until the `*/` appears. In other words, the statement just sets `y` to the value of `x` and doesn't even look at `p`. Rewriting this statement as

```
y = x / *p /* p points at the divisor */;
```

or even

```
y = x/(*p) /* p points at the divisor */;
```

would cause it to do the division the comment suggests.

This sort of near-ambiguity can cause trouble in other contexts. For example, older versions of C use `=+` to mean what present versions mean by `+=`. Such a compiler will treat

```
a=-1;
```

as meaning the same thing as

```
a == 1;
```

or

```
a = a - 1;
```

This will surprise a programmer who intended

```
a = -1;
```

On the other hand, compilers for these older versions of C would interpret

```
a=/*b;
```

as

```
a =/ * b ;
```

even though the `/*` looks like a comment.

1.4. Exceptions

Compound assignment operators such as `=+` are really multiple tokens. Thus,

```
a + /* strange */ = 1
```

means the same as

```
a += 1
```

These operators are the only cases in which things that look like single tokens are really multiple tokens. In particular,

```
p - > a
```

is illegal. It is *not* a synonym for

```
p -> a
```

As another example, the `>>` operator is a single token, so `>>=` is made up of two tokens, not three.

On the other hand, those older compilers that still accept `=+` as a synonym for `+=` treat `=+` as a single token.

1.5. Strings and Characters

Single and double quotes mean very different things in C, and there are some contexts in which confusing them will result in surprises rather than error messages.

A character enclosed in single quotes is just another way of writing an integer. The integer is the one that corresponds to the given character in the implementation's collating sequence. Thus, in an ASCII implementation, 'a' means exactly the same thing as 0141 or 97. A string enclosed in double quotes, on the other hand, is a short-hand way of writing a pointer to a nameless array that has been initialized with the characters between the quotes and an extra character whose binary value is zero.

The following two program fragments are equivalent:

```
printf ("Hello world\n");

char hello[] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\n', 0};
printf (hello);
```

Using a pointer instead of an integer (or vice versa) will often cause a warning message, so using double quotes instead of single quotes (or vice versa) is usually caught. The major exception is in function calls, where most compilers do not check argument types. Thus, saying

```
printf ('\n');
```

instead of

```
printf ("\\n");
```

will usually result in a surprise at run time.

Because an integer is usually large enough to hold several characters, some C compilers permit multiple characters in a character constant. This means that writing 'yes' instead of "yes" may well go undetected. The latter means "the address of the first of four consecutive memory locations containing y, e, s, and a null character, respectively." The former means "an integer that is composed of the values of the characters y, e, and s in some implementation-defined manner." Any similarity between these two quantities is purely coincidental.

2. Syntactic Pitfalls

To understand a C program, it is not enough to understand the tokens that make it up. One must also understand how the tokens combine to form declarations, expressions, statements, and programs. While these combinations are usually well-defined, the definitions are sometimes counter-intuitive or confusing.

In this section, we look at some syntactic constructions that are less than obvious.

2.1. Understanding Declarations

I once talked to someone who was writing a C program that was going to run stand-alone in a small microprocessor. When this machine was switched on, the hardware would call the subroutine whose address was stored in location 0.

In order to simulate turning power on, we had to devise a C statement that would call this subroutine explicitly. After some thought, we came up with the following:

```
(* (void(*)()) 0)();
```

Expressions like these strike terror into the hearts of C programmers. They needn't, though, because they can usually be constructed quite easily with the help of a single, simple rule: *declare it the way you use it*.

Every C variable declaration has two parts: a type and a list of stylized expressions that are expected to evaluate to that type. The simplest such expression is a variable:

```
float f, g;
```

indicates that the expressions `f` and `g`, when evaluated, will be of type `float`. Because the thing declared is an expression, parentheses may be used freely:

```
float ((f));
```

means that `((f))` evaluates to a `float` and therefore, by inference, that `f` is also a `float`.

Similar logic applies to function and pointer types. For example,

```
float ff();
```

means that the expression `ff()` is a `float`, and therefore that `ff` is a function that returns a `float`. Analogously,

```
float *pf;
```

means that `*pf` is a `float` and therefore that `pf` is a pointer to a `float`.

These forms combine in declarations the same way they do in expressions. Thus

```
float *g(), (*h());
```

says that `*g()` and `(*h())` are `float` expressions. Since `()` binds more tightly than `*`, `*g()` means the same thing as `*(g())`: `g` is a function that returns a pointer to a `float`, and `h` is a pointer to a function that returns a `float`.

Once we know how to declare a variable of a given type, it is easy to write a cast for that type: just remove the variable name and the semicolon from the declaration and enclose the whole thing in parentheses. Thus, since

```
float *g();
```

declares `g` to be a function returning a pointer to a `float`, `(float *())` is a cast to that type.

Armed with this knowledge, we are now prepared to tackle `(*void(*)())()`. We can analyze this statement in two parts. First, suppose that we have a variable `fp` that contains a function pointer and we want to call the function to which `fp` points. That is done this way:

```
(*fp)();
```

If `fp` is a pointer to a function, `*fp` is the function itself, so `(*fp)()` is the way to invoke it. The parentheses in `(*fp)` are essential because the expression would otherwise be interpreted as `*(fp())`. We have now reduced the problem to that of finding an appropriate expression to replace `fp`.

This problem is the second part of our analysis. If C could read our mind about types, we could write:

```
(*0)();
```

This doesn't work because the `*` operator insists on having a pointer as its operand. Furthermore, the operand must be a pointer to a function so that the result of `*` can be called. Thus, we need to cast 0 into a type loosely described as "pointer to function returning void."

If `fp` is a pointer to a function returning void, then `(*fp)()` is a void value, and its declaration would look like this:

```
void (*fp)();
```

Thus, we could write:

```
void (*fp)();  
(*fp)();
```

at the cost of declaring a dummy variable. But once we know how to declare the variable, we know how to cast a constant to that type: just drop the name from the variable declaration. Thus, we cast 0 to a "pointer to function returning void" by saying:

```
(void(*)())0
```

and we can now replace `fp` by `(void(*)())0`:

```
(* (void(*)())0)();
```

The semicolon on the end turns the expression into a statement.

At the time we tackled this problem, there was no such thing as a `typedef` declaration. Using it, we could have solved the problem more clearly:

```
typedef void (*funcptr)();
(* (funcptr) 0)();
```

2.2. Operators Don't Always Have the Precedence You Want

Suppose that the manifest constant `FLAG` is an integer with exactly one bit turned on in its binary representation (in other words, a power of two), and you want to test whether the integer variable `flags` has that bit turned on. The usual way to write this is:

```
if (flags & FLAG) ...
```

The meaning of this is plain to most C programmers: an `if` statement tests whether the expression in the parentheses evaluates to 0 or not. It might be nice to make this test more explicit for documentation purposes:

```
if (flags & FLAG != 0) ...
```

The statement is now easier to understand. It is also wrong, because `!=` binds more tightly than `&`, so the interpretation is now:

```
if (flags & (FLAG != 0)) ...
```

This will work (by coincidence) if `FLAG` is 1 or 0 (!), but not for any other power of two.*

Suppose you have two integer variables, `h` and `l`, whose values are between 0 and 15 inclusive, and you want to set `r` to an 8-bit value whose low-order bits are those of `l` and whose high-order bits are those of `h`. The natural way to do this is to write:

```
r = h<<4 + l;
```

Unfortunately, this is wrong. Addition binds more tightly than shifting, so this example is equivalent to

```
r = h << (4 + l);
```

Here are two ways to get it right:

```
r = (h << 4) + l;
r = h << 4 | l;
```

One way to avoid these problems is to parenthesize everything, but expressions with too many parentheses are hard to understand, so it is probably useful to try to remember the precedence levels in C.

Unfortunately, there are fifteen of them, so this is not always easy to do. It can be made easier, though, by classifying them into groups.

The operators that bind the most tightly are the ones that aren't really operators: subscripting, function calls, and structure selection. These all associate to the left.

Next come the unary operators. These have the highest precedence of any of the true operators. Because function calls bind more tightly than unary operators, you must write `(*p)()` to call a function pointed to by `p`; `*p()` implies that `p` is a function that returns a pointer. Casts are unary operators and have the same precedence as any other unary operator. Unary operators are right-associative, so `*p++` is

* Recall that the result of `!=` is always either 1 or 0.

interpreted as `*(p++)` and not as `(*p)++`.

Next come the true binary operators. The arithmetic operators have the highest precedence, then the shift operators, the relational operators, the logical operators, the assignment operators, and finally the conditional operator. The two most important things to keep in mind are:

1. Every logical operator has lower precedence than every relational operator.
2. The shift operators bind more tightly than the relational operators but less tightly than the arithmetic operators.

Within the various operator classes, there are few surprises. Multiplication, division, and remainder have the same precedence, addition and subtraction have the same precedence, and the two shift operators have the same precedence.

One small surprise is that the six relational operators do not all have the same precedence: `==` and `!=` bind less tightly than the other relational operators. This allows us, for instance, to see if `a` and `b` are in the same relative order as `c` and `d` by the expression

```
a < b == c < d
```

Within the logical operators, no two have the same precedence. The bitwise operators all bind more tightly than the sequential operators, each *and* operator binds more tightly than the corresponding *or* operator, and the bitwise *exclusive or* operator (`^`) falls between bitwise *and* and bitwise *or*.

The ternary conditional operator has lower precedence than any we have mentioned so far. This permits the selection expression to contain logical combinations of relational operators, as in

```
z = a < b && b < c ? d : e
```

This example also shows that it makes sense for assignment to have a lower precedence than the conditional operator. Moreover, all the compound assignment operators have the same precedence and they all group right to left, so that

```
a = b = c
```

means the same as

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

Lowest of all is the comma operator. This is easy to remember because the comma is often used as a substitute for the semicolon when an expression is required instead of a statement.

Assignment is another operator often involved in precedence mixups. Consider, for example, the following loop intended to copy one file to another:

```
while (c=getc(in) != EOF)
    putc(c,out);
```

The way the expression in the `while` statement is written makes it look like `c` should be assigned the value of `getc(in)` and then compared with `EOF` to terminate the loop. Unhappily, assignment has lower precedence than any comparison operator, so the value of `c` will be the result of comparing `getc(in)`, the value of which is then discarded, and `EOF`. Thus, the “copy” of the file will consist of a stream of bytes whose value is 1.

It is not too hard to see that the example above should be written:

```
while ((c=getc(in)) != EOF)
    putc(c,out);
```

However, errors of this sort can be hard to spot in more complicated expressions. For example, several versions of the `lint` program distributed with the UNIX® system have the following erroneous line:

```
if( (t=BTYPE(pt1->aty)==STRTY) || t==UNIONTY ) {
```

This was intended to assign a value to `t` and then see if `t` is equal to `STRTY` or `UNIONTY`. The actual

effect is quite different.*

The precedence of the C logical operators comes about for historical reasons. B, the predecessor of C, had logical operators that corresponded roughly to C's & and | operators. Although they were defined to act on bits, the compiler would treat them as && and || if they were in a conditional context. When the two usages were split apart in C, it was deemed too dangerous to change the precedence much.**

2.3. Watch Those Semicolons!

An extra semicolon in a C program usually makes little difference: either it is a null statement, which has no effect, or it elicits a diagnostic message from the compiler, which makes it easy to remove. One important exception is after an `if` or `while` clause, which must be followed by exactly one statement. Consider this example:

```
if (x[i] > big);  
    big = x[i];
```

The semicolon on the first line will not upset the compiler, but this program fragment means something quite different from:

```
if (x[i] > big)  
    big = x[i];
```

The first one is equivalent to:

```
if (x[i] > big) { }  
big = x[i];
```

which is, of course, equivalent to:

```
big = x[i];
```

(unless `x`, `i`, or `big` is a macro with side effects).

Another place that a semicolon can make a big difference is at the end of a declaration just before a function definition. Consider the following fragment:

```
struct foo {  
    int x;  
}  
  
f()  
{  
    . . .  
}
```

There is a semicolon missing between the first } and the `f` that immediately follows it. The effect of this is to declare that the function `f` returns a `struct foo`, which is defined as part of this declaration. If the semicolon were present, `f` would be defined by default as returning an integer.†

2.4. The Switch Statement

C is unusual in that the cases in its `switch` statement can flow into each other. Consider, for example, the following program fragments in C and Pascal:

* Thanks to Guy Harris for pointing this out to me.

** Dennis Ritchie and Steve Johnson both pointed this out to me.

† Thanks to an anonymous benefactor for this one.

```
switch (color) {
    case 1: printf ("red");
              break;
    case 2: printf ("yellow");
              break;
    case 3: printf ("blue");
              break;
}

case color of
1:  write ('red');
2:  write ('yellow');
3:  write ('blue')
end
```

Both these program fragments do the same thing: print red, yellow, or blue (without starting a new line), depending on whether the variable `color` is 1, 2, or 3. The program fragments are exactly analogous, with one exception: the Pascal program does not have any part that corresponds to the C `break` statement. The reason for that is that case labels in C behave as true labels: control can flow unimpeded right through a case label.

Looking at it another way, suppose the C fragment looked more like the Pascal fragment:

```
switch (color) {
    case 1: printf ("red");
    case 2: printf ("yellow");
    case 3: printf ("blue");
}
```

and suppose further that `color` were equal to 2. Then, the program would print yellowblue, because control would pass naturally from the second `printf` call to the statement after it.

This is both a strength and a weakness of C `switch` statements. It is a weakness because leaving out a `break` statement is easy to do, and often gives rise to obscure program misbehavior. It is a strength because by leaving out a `break` statement deliberately, one can readily express a control structure that is inconvenient to implement otherwise. Specifically, in large `switch` statements, one often finds that the processing for one of the cases reduces to some other case after a relatively small amount of special handling.

For example, consider a program that is an interpreter for some kind of imaginary machine. Such a program might contain a `switch` statement to handle each of the various operation codes. On such a machine, it is often true that a subtract operation is identical to an add operation after the sign of the second operand has been inverted. Thus, it is nice to be able to write something like this:

```
case SUBTRACT:
    opnd2 = -opnd2;
    /* no break */
case ADD:
    . . .
```

As another example, consider the part of a compiler that skips white space while looking for a token. Here, one would want to treat spaces, tabs, and newlines identically except that a newline should cause a line counter to be incremented:

```
case '\n':
    linecount++;
    /* no break */
case '\t':
case ' ':
    . . .
```

2.5. Calling Functions

Unlike some other programming languages, C requires a function call to have an argument list, even if there are no arguments. Thus, if `f` is a function,

```
f();
```

is a statement that calls the function, but

```
f;
```

does nothing at all. More precisely, it evaluates the address of the function, but does not call it.*

2.6. The Dangling else Problem

We would be remiss in leaving any discussion of syntactic pitfalls without mentioning this one. Although it is not unique to C, it has bitten C programmers with many years of experience.

Consider the following program fragment:

```
if (x == 0)
    if (y == 0) error();
else {
    z = x + y;
    f (&z);
}
```

The programmer's intention for this fragment is that there should be two main cases: $x=0$ and $x\neq 0$. In the first case, the fragment should do nothing at all unless $y=0$, in which case it should call `error`. In the second case, the program should set $z=x+y$ and then call `f` with the address of `z` as its argument.

However, the program fragment actually does something quite different. The reason is the rule that an `else` is always associated with the closest unmatched `if`. If we were to indent this fragment the way it is actually executed, it would look like this:

```
if (x == 0) {
    if (y == 0)
        error();
    else {
        z = x + y;
        f (&z);
    }
}
```

In other words, nothing at all will happen if $x\neq 0$. To get the effect implied by the indentation of the original example, write:

* Thanks to Richard Stevens for pointing this out.

```
if (x == 0) {
    if (y == 0)
        error();
} else {
    z = x + y;
    f (&z);
}
```

3. Linkage

A C program may consist of several parts that are compiled separately and then bound together by a program usually called a *linker*, *linkage editor*, or *loader*. Because the compiler normally sees only one file at a time, it cannot detect errors whose recognition would require knowledge of several source program files at once.

In this section, we look at some errors of that type. Some C implementations, but not all, have a program called *lint* that catches many of these errors. It is impossible to overemphasize the importance of using such a program if it is available.

3.1. You Must Check External Types Yourself

Suppose you have a C program divided into two files. One file contains the declaration:

```
int n;
```

and the other contains the declaration:

```
long n;
```

This is not a valid C program, because the same external name is declared with two different types in the two files. However, many implementations will not detect this error, because the compiler does not know about the contents of either of the two files while it is compiling the other. Thus, the job of checking type consistency can only be done by the linker (or some utility program like *lint*); if the operating system has a linker that doesn't know about data types, there is little the C compiler can do to force it.

What actually happens when this program is run? There are many possibilities:

1. The implementation is clever enough to detect the type clash. One would then expect to see a diagnostic message explaining that the type of *n* was given differently in two different files.
2. You are using an implementation in which *int* and *long* are really the same type. This is typically true of machines in which 32-bit arithmetic comes most naturally. In this case, your program will probably work as if you had said *long* (or *int*) in both declarations. This would be a good example of a program that works only by coincidence.
3. The two instances of *n* require different amounts of storage, but they happen to share storage in such a way that the values assigned to one are valid for the other. This might happen, for example, if the linker arranged for the *int* to share storage with the low-order part of the *long*. Whether or not this happens is obviously machine- and system-dependent. This is an even better example of a program that works only by coincidence.
4. The two instances of *n* share storage in such a way that assigning a value to one has the effect of apparently assigning a different value to the other. In this case, the program will probably fail.

Another example of this sort of thing happens surprisingly often. One file of a program will contain a declaration like:

```
char filename[] = "/etc/passwd";
```

and another will contain this declaration:

```
char *filename;
```

Although arrays and pointers behave very similarly in some contexts, *they are not the same*. In the first declaration, `filename` is the name of an array of characters. Although using the name will generate a pointer to the first element of that array, that pointer is generated as needed and not actually kept around. In the second declaration, `filename` is the name of a pointer. That pointer points wherever the programmer makes it point. If the programmer doesn't give it a value, it will have a zero (null) value by default.

The two declarations of `filename` use storage in different ways; they cannot coexist.

One way to avoid type clashes of this sort is to use a tool like *lint* if it is available. In order to be able to check for type clashes between separately compiled parts of a program, some program must be able to see all the parts at once. The typical compiler does not do this, but *lint* does.

Another way to avoid these problems is to put external declarations into `include` files. That way, the type of an external object only appears once.*

4. Semantic Pitfalls

A sentence can be perfectly spelled and written with impeccable grammar and still be meaningless. In this section, we will look at ways of writing programs that look like they mean one thing but actually mean something quite different.

We will also discuss contexts in which things that look reasonable on the surface actually give undefined results. We will limit ourselves here to things that are not guaranteed to work on any C implementation. We will leave those that might work on some implementations but not others until section 7, which looks at portability problems.

4.1. Expression Evaluation Sequence

Some C operators always evaluate their operands in a known, specified order. Others don't. Consider, for instance, the following expression:

```
a < b && c < d
```

The language definition states that `a<b` will be evaluated first. If `a` is indeed less than `b`, `c<d` must then be evaluated to determine the value of the whole expression. On the other hand, if `a` is greater than or equal to `b`, then `c<d` is not evaluated at all.

To evaluate `a<b`, on the other hand, the compiler may evaluate either `a` or `b` first. On some machines, it may even evaluate them in parallel.

Only the four C operators `&&`, `||`, `? :`, and `,` specify an order of evaluation. `&&` and `||` evaluate the left operand first, and the right operand only if necessary. The `? :` operator takes three operands: `a?b:c` evaluates `a` first, and then evaluates either `b` or `c`, depending on the value of `a`. The `,` operator evaluates its left operand and discards its value, then evaluates its right operand.†

All other C operators evaluate their operands in undefined order. In particular, the assignment operators do not make any guarantees about evaluation order.

For this reason, the following way of copying the first `n` elements of array `x` to array `y` doesn't work:

```
i = 0;
while (i < n)
    y[i] = x[i++];
```

The trouble is that there is no guarantee that the address of `y[i]` will be evaluated before `i` is incremented.

* Some C compilers insist that there must be exactly one definition of an external object, although there may be many declarations. When using such a compiler, it may be easiest to put a declaration in an `include` file and a definition in some other place. This means that the type of each external object appears twice, but that is better than having it appear more than two times.

† Commas that separate function arguments are not comma operators. For example, `x` and `y` are fetched in undefined order in `f(x, y)`, but not in `g((x, y))`. In the latter example, `g` has one argument. The value of that argument is determined by evaluating `x`, discarding its value, and then evaluating `y`.

On some implementations, it will; on others, it won't. This similar version fails for the same reason:

```
i = 0;
while (i < n)
    y[i++] = x[i];
```

On the other hand, this one will work fine:

```
i = 0;
while (i < n) {
    y[i] = x[i];
    i++;
}
```

This can, of course, be abbreviated:

```
for (i = 0; i < n; i++)
    y[i] = x[i];
```

4.2. The `&&`, `||`, and `!` Operators

C has two classes of logical operators that are occasionally interchangeable: the bitwise operators `&`, `|`, and `~`, and the logical operators `&&`, `||`, and `!`. A programmer who substitutes one of these operators for the corresponding operator from the other class may be in for a surprise: the program may appear to work correctly after such an interchange but may actually be working only by coincidence.

The `&`, `|`, and `~` operators treat their operands as a sequence of bits and work on each bit separately. For example, $10 \& 12$ is 8 (1000), because `&` looks at the binary representations of 10 (1010) and 12 (1100) and produces a result that has a bit turned on for each bit that is on in the same position in both operands. Similarly, $10 | 12$ is 14 (1110) and ~ 10 is -11 (11...110101), at least on a 2's complement machine.

The `&&`, `||`, and `!` operators, on the other hand, treat their arguments as if they are either “true” or “false,” with the convention that 0 represents “false” and any other value represents “true.” These operators return 1 for “true” and 0 for “false,” and the `&&` and `||` operators do not even evaluate their right-hand operands if their results can be determined from their left-hand operands.

Thus $!10$ is zero, because 10 is nonzero, $10 \&& 12$ is 1, because both 10 and 12 are nonzero, and $10 || 12$ is also 1, because 10 is nonzero. Moreover, 12 is not even evaluated in the latter expression, nor is `f()` in $10 || f()$.

Consider the following program fragment to look for a particular element in a table:

```
i = 0;
while (i < tabsz && tab[i] != x)
    i++;
```

The idea behind this loop is that if `i` is equal to `tabsz` when the loop terminates, then the element sought was not found. Otherwise, `i` contains the element’s index.

Suppose that the `&&` were inadvertently replaced by `&` in this example. Then the loop would probably still appear to work, but would do so only because of two lucky breaks.

The first is that both comparisons in this example are of a sort that yield 0 if the condition is false and 1 if the condition is true. As long as `x` and `y` are both 1 or 0, `x&y` and `x&&y` will always have the same value. However, if one of the comparisons were to be replaced by one that uses some non-zero value other than 1 to represent “true,” then the loop would stop working.

The second lucky break is that looking just one element off the end of an array is usually harmless, provided that the program doesn’t change that element. The modified program looks past the end of the array because `&`, unlike `&&`, must always evaluate both of its operands. Thus in the last iteration of the loop, the value of `tab[i]` will be fetched even though `i` is equal to `tabsz`. If `tabsz` is the number of elements in `tab`, this will fetch a non-existent element of `tab`.

4.3. Subscripts Start from Zero

In most languages, an array with n elements normally has those elements numbered with subscripts ranging from 1 to n inclusive. Not so in C.

A C array with n elements does not have an element with a subscript of n , as the elements are numbered from 0 through $n-1$. Because of this, programmers coming from other languages must be especially careful when using arrays:

```
int i, a[10];
for (i=1; i<=10; i++)
    a[i] = 0;
```

This example, intended to set the elements of a to zero, had an unexpected side effect. Because the comparison in the `for` statement was `i<=10` instead of `i<10`, the non-existent element number 10 of a was set to zero, thus clobbering the word that followed a in memory. The compiler on which this program was run allocates memory for users' variables in decreasing memory locations, so the word after a turned out to be i . Setting i to zero made the loop into an infinite loop.

4.4. C Doesn't Always Cast Actual Parameters

The following simple program fragment fails for two reasons:

```
double s;
s = sqrt (2);
printf ("%g\n", s);
```

The first reason is that `sqrt` expects a `double` value as its argument and it isn't getting one. The second is that it returns a `double` result but isn't declared as such. One way to correct it is:

```
double s, sqrt();
s = sqrt (2.0);
printf ("%g\n", s);
```

C has two simple rules that control conversion of function arguments: (1) integer values shorter than an `int` are converted to `int`; (2) floating-point values shorter than a `double` are converted to `double`. All other values are left unconverted. *It is the programmer's responsibility to ensure that the arguments to a function are of the right type.*

Therefore, a programmer who uses a function like `sqrt`, whose parameter is a `double`, must be careful to pass arguments that are of `float` or `double` type only. The constant 2 is an `int` and is therefore of the wrong type.

When the value of a function is used in an expression, that value is automatically cast to an appropriate type. However, the compiler must know the actual type returned by the function in order to be able to do this. Functions used without further declaration are assumed to return an `int`, so declarations for such functions are unnecessary. However, `sqrt` returns a `double`, so it must be declared as such before it can be used successfully.

In practice, C implementations generally provide a file that can be brought in with an `include` statement that contains declarations for library functions like `sqrt`, but writing declarations is still necessary for programmers who write their own functions – in other words, for anyone who writes non-trivial C programs.

Here is a more spectacular example:

```
main()
{
    int i;
    char c;
    for (i=0; i<5; i++) {
        scanf ("%d", &c);
        printf ("%d ", i);
    }
    printf ("\n");
}
```

Ostensibly, this program reads five numbers from its standard input and writes 0 1 2 3 4 on its standard output. In fact, it doesn't always do that. On one compiler, for example, its output is 0 0 0 0 0 1 2 3 4.

Why? The key is the declaration of `c` as a `char` rather than as an `int`. When you ask `scanf` to read an integer, it expects a pointer to an integer. What it gets in this case is a pointer to a character. `Scanf` has no way to tell that it didn't get what it expected: it treats its input as an integer pointer and stores an integer there. Since an integer takes up more memory than a character, this steps on some of the memory near `c`.

Exactly what is near `c` is the compiler's business; in this case it turned out to be the low-order part of `i`. Therefore, each time a value was read for `c`, it reset `i` to zero. When the program finally reached end of file, `scanf` stopped trying to put new values into `c`, so `i` could be incremented normally to end the loop.

4.5. Pointers are not Arrays

C programs often adopt the convention that a character string is stored as an array of characters, followed by a null character. Suppose we have two such strings `s` and `t`, and we want to concatenate them into a single string `r`. To do this, we have the usual library functions `strcpy` and `strcat`. The following obvious method doesn't work:

```
char *r;
strcpy (r, s);
strcat (r, t);
```

The reason it doesn't work is that `r` is not initialized to point anywhere. Although `r` is potentially capable of identifying an area of memory, *that area doesn't exist until you allocate it*.

Let's try again, allocating some memory for `r`:

```
char r[100];
strcpy (r, s);
strcat (r, t);
```

This now works as long as the strings pointed to by `s` and `t` aren't too big. Unfortunately, C requires us to state the size of an array as a constant, so there is no way to be certain that `r` will be big enough. However, most C implementations have a library function called `malloc` that takes a number and allocates enough memory for that many characters. There is also usually a function called `strlen` that tells how many characters are in a string. It might seem, therefore, that we could write:

```
char *r, *malloc();
r = malloc (strlen(s) + strlen(t));
strcpy (r, s);
strcat (r, t);
```

This example, however, fails for two reasons. First, `malloc` might run out of memory, an event that it generally signals by quietly returning a null pointer.

Second, and much more important, is that the call to `malloc` doesn't allocate quite enough memory. Recall the convention that a string is terminated by a null character. The `strlen` function returns the

number of characters in the argument string, *excluding* the null character at the end. Therefore, if `strlen(s)` is n , `s` really requires $n+1$ characters to contain it. We must therefore allocate one extra character for `r`. After doing this and checking that `malloc` worked, we get:

```
char *r, *malloc();
r = malloc (strlen(s) + strlen(t) + 1);
if (!r) {
    complain();
    exit (1);
}
strcpy (r, s);
strcat (r, t);
```

4.6. Eschew Synecdoche

A synecdoche (sin-ECK-duh-key) is a literary device, somewhat like a simile or a metaphor, in which, according to the Oxford English Dictionary, “a more comprehensive term is used for a less comprehensive or vice versa; as whole for part or part for whole, genus for species or species for genus, etc.”

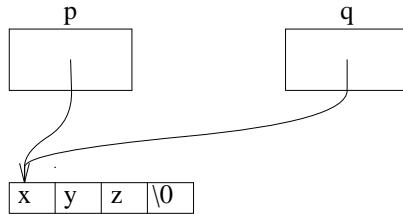
This exactly describes the common C pitfall of confusing a pointer with the data to which it points. This is most common for character strings. For instance:

```
char *p, *q;
p = "xyz";
```

It is important to understand that while it is sometimes useful to think of the value of `p` as the string `xyz` after the assignment, this is not really true. Instead, the value of `p` is a *pointer* to the 0th element of an array of four characters, whose values are '`x`', '`y`', '`z`', and '`\0`'. Thus, if we now execute

```
q = p;
```

`p` and `q` are now two pointers to the same part of memory. The characters in that memory did not get copied by the assignment. The situation now looks like this:



The thing to remember is that *copying a pointer does not copy the thing it points to*.

Thus, if after this we were to execute

```
q[1] = 'Y';
```

`q` would point to memory containing the string `xyz`. So would `p`, because `p` and `q` point to the same memory.

4.7. The Null Pointer is Not the Null String

The result of converting an integer to a pointer is implementation-dependent, with one important exception. That exception is the constant 0, which is guaranteed to be converted to a pointer that is unequal to any valid pointer. For documentation, this value is often given symbolically:

```
#define NULL 0
```

but the effect is the same. The important thing to remember about 0 when used as a pointer is that *it must never be dereferenced*. In other words, when you have assigned 0 to a pointer variable, you must not ask

what is in the memory it points to. It is valid to write:

```
if (p == (char *) 0) ...
```

but it is not valid to write:

```
if (strcmp (p, (char *) 0) == 0) ...
```

because `strcmp` always looks at the memory addressed by its arguments.

If `p` is a null pointer, it is not even valid to say:

```
printf (p);
```

or

```
printf ("%s", p);
```

4.8. Integer Overflow

The C language definition is very specific about what happens when an integer operation overflows or underflows.

If either operand is unsigned, the result is unsigned, and is defined to be modulo 2^n , where n is the word size. If both operands are signed, the result is *undefined*.

Suppose, for example, that `a` and `b` are two integer variables, known to be non-negative, and you want to test whether `a+b` might overflow. One obvious way to do it looks something like this:

```
if (a + b < 0)
    complain();
```

In general, this does not work.

The point is that once `a+b` has overflowed, all bets are off as to what the result will be. For example, on some machines, an addition operation sets an internal register to one of four states: positive, negative, zero, or overflow. On such a machine, the compiler would have every right to implement the example given above by adding `a` and `b` and checking whether this internal register was in negative state afterwards. If the operation overflowed, the register would be in overflow state, and the test would fail.

One correct way of doing this particular test relies on the fact that unsigned arithmetic is well-defined for all values, as are the conversions between signed and unsigned values:

```
if ((int) ((unsigned) a + (unsigned) b) < 0)
    complain();
```

4.9. Shift Operators

Two questions seem to cause trouble for people who use shift operators:

1. In a right shift, are vacated bits filled with zeroes or copies of the sign bit?
2. What values are permitted for the shift count?

The answer to the first question is simple but sometimes implementation-dependent. If the item being shifted is unsigned, zeroes are shifted in. If the item is signed, the implementation is permitted to fill vacated bit positions either with zeroes or with copies of the sign bit. If you care about vacated bits in a right shift, declare the variable in question as `unsigned`. You are then entitled to assume that vacated bits will be set to zero.

The answer to the second question is also simple: if the item being shifted is n bits long, then the shift count must be greater than or equal to zero and *strictly* less than n . Thus, it is not possible to shift all the bits out of a value in a single operation.

For example, if an `int` is 32 bits, and `n` is an `int`, it is legal to write `n<<31` and `n<<0` but not `n<<32` or `n<<-1`.

Note that a right shift of a signed integer is generally not equivalent to division by a power of two,

even if the implementation copies the sign into vacated bits. To prove this, consider that the value of $(-1) >> 1$ cannot possibly be zero.

5. Library Functions

Every useful C program must use library functions, because there is no way of doing input or output built into the language. In this section, we look at some cases where some widely-available library functions behave in ways that the programmer might not expect.

5.1. Getc Returns an Integer

Consider the following program:

```
#include <stdio.h>

main()
{
    char c;

    while ((c = getchar()) != EOF)
        putchar (c);
}
```

This program looks like it should copy its standard input to its standard output. In fact, it doesn't quite do this.

The reason is that `c` is declared as a character rather than as an integer. This means that it is impossible for `c` to hold every possible character as well as EOF.

Thus there are two possibilities. Either some legitimate input character will cause `c` to take on the same value as EOF, or it will be impossible for `c` to have the value EOF at all. In the former case, the program will stop copying in the middle of certain files. In the latter case, the program will go into an infinite loop.

Actually, there is a third case: the program may work by coincidence. The C Reference Manual defines the result of the expression

```
((c = getchar()) != EOF)
```

quite rigorously. Section 6.1 states:

When a longer integer is converted to a shorter or to a `char`, it is truncated on the left; excess bits are simply discarded.

Section 7.14 states:

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

The combined effect of these two sections is to require that the result of `getchar` be truncated to a character value by discarding the high-order bits, and that this truncated value then be compared with EOF. As part of this comparison, the value of `c` must be extended to an integer, either by padding on the left with zero bits or by sign extension, as appropriate.

However, some compilers do not implement this expression correctly. They properly assign the low-order bits of the value of `getchar` to `c`. However, instead of then comparing `c` to EOF, they compare the entire value of `getchar`! A compiler that does this will make the sample program shown above appear to work "correctly."

5.2. Buffered Output and Memory Allocation

When a program produces output, how important is it that a human be able to see that output immediately? It depends on the program.

For example, if the output is going to a terminal and is asking the person sitting at that terminal to answer a question, it is crucial that the person see the output in order to be able to know what to type. On the other hand, if the output is going to a file, from where it will eventually be sent to a line printer, it is only important that all the output get there eventually.

It is often more expensive to arrange for output to appear immediately than it is to save it up for a while and write it later on in a large chunk. For this reason, C implementations typically afford the programmer some control over how much output is to be produced before it is actually written.

That control is often vested in a library function called *setbuf*. If *buf* is a character array of appropriate size, then

```
setbuf (stdout, buf);
```

tells the I/O library that all output written to *stdout* should henceforth use *buf* as an output buffer, and that output directed to *stdout* should not actually be written until *buf* becomes full or until the programmer directs it to be written by calling *fflush*. The appropriate size for such a buffer is defined as *BUFSIZ* in *<stdio.h>*.

Thus, the following program illustrates the obvious way to use *setbuf* in a program that copies its standard input to its standard output:

```
#include <stdio.h>

main()
{
    int c;

    char buf[BUFSIZ];
    setbuf (stdout, buf);

    while ((c = getchar()) != EOF)
        putchar (c);
}
```

Unfortunately, this program is wrong, for a subtle reason.

To see where the trouble lies, ask when the buffer is flushed for the last time. Answer: after the main program has finished, as part of the cleaning up that the library does before handing control back to the operating system. But by that time, the buffer has already been freed!

There are two ways to prevent this sort of trouble.

First, make the buffer static, either by declaring it explicitly as static:

```
static char buf[BUFSIZ];
```

or by moving the declaration outside the main program entirely.

Another possibility is to allocate the buffer dynamically and never free it:

```
char *malloc();
setbuf (stdout, malloc (BUFSIZ));
```

Note that in this latter case, it is unnecessary to check if *malloc* was successful, because if it fails it will return a null pointer. A null pointer is an acceptable second argument to *setbuf*; it requests that *stdout* be unbuffered. This will work slowly, but it will work.

6. The Preprocessor

The programs we run are not the programs we write: they are first transformed by the C preprocessor. The preprocessor gives us a way of abbreviating things that is important for two major reasons (and several minor ones).

First, we may want to be able to change all instances of a particular quantity, such as the size of a table, by changing one number and recompiling the program.*

Second, we may want to define things that appear to be functions but do not have the execution overhead normally associated with a function call. For example, *getchar* and *putchar* are usually implemented as macros to avoid having to call a function for each character of input or output.

6.1. Macros are not Functions

Because macros can be made to appear almost as if they were functions, programmers are sometimes tempted to regard them as truly equivalent. Thus, one sees things like this:

```
#define max(a,b) ((a)>(b)?(a):(b))
```

Notice all the parentheses in the macro body. They defend against the possibility that *a* or *b* might be expressions that contain operators of lower precedence than *>*.

The main problem, though, with defining things like *max* as macros is that an operand that is used twice may be evaluated twice. Thus, in this example, if *a* is greater than *b*, *a* will be evaluated twice: once during the comparison, and again to calculate the value yielded by *max*.

Not only can this be inefficient, it can also be wrong:

```
biggest = x[0];
i = 1;
while (i < n)
    biggest = max (biggest, x[i++]);
```

This would work fine if *max* were a true function, but fails with *max* a macro. Suppose, for example, that *x[0]* is 2, *x[1]* is 3, and *x[2]* is 1. Look at what happens during the first iteration of the loop. The assignment statement expands into:

```
biggest = ((biggest)>(x[i++])?(biggest):(x[i++]));
```

First, *biggest* is compared to *x[i++]*. Since *i* is 1 and *x[1]* is 3, the relation is false. As a side effect, *i* is incremented to 2.

Because the relation is false, the value of *x[i++]* is now assigned to *biggest*. However, *i* is now 2, so the value assigned to *biggest* is the value of *x[2]*, which is 1.

One way around these worries is to ensure that the arguments to the *max* macro do not have any side effects:

```
biggest = x[0];
for (i = 1; i < n; i++)
    biggest = max (biggest, x[i]);
```

Here is another example of the hazards of mixing side effects and macros. This is the definition of the *putc* macro from *<stdio.h>* in the Eighth Edition of the Unix system:

```
#define putc(x,p) (--(p)->_cnt>=0?(*p)->_ptr++=(x):_flsbuf(x,p))
```

The first argument to *putc* is a character to be written to a file; the second argument is a pointer to an internal data structure that describes the file. Notice that the first argument, which could easily be something like **z++*, is carefully evaluated only once, even though it appears in two separate places in the macro body, while the second argument is evaluated twice (in the macro body, *x* appears twice, but since

* The preprocessor also makes it easy to group such *manifest constants* together to make them easier to find.

the two occurrences are on opposite sides of a : operator, exactly one of them will be evaluated in any single instance of `putc`). Since it is unusual for the file argument to `putc` to have side effects, this rarely causes trouble. Nevertheless, it is documented in the user's manual: "Because it is implemented as a macro, `putc` treats a *stream* argument with side effects improperly. In particular, `putc(c, *f++)` doesn't work sensibly." Notice that `putc(*c++, f)` works fine in this implementation.

Some C implementations are less careful. For instance, not everyone handles `putc(*c++, f)` correctly. As another example, consider the `toupper` function that appears in many C libraries. It translates a lower-case letter to the corresponding upper-case letter while leaving other characters unchanged. If we assume that all the lower-case letters and all the upper-case letters are contiguous (with a possible gap between the cases), we get the following function:

```
toupper(c)
{
    if (c >= 'a' && c <= 'z')
        c += 'A' - 'a';
    return c;
}
```

In most C implementations, the subroutine call overhead is much longer than the actual calculations, so the implementor is tempted to make it a macro:

```
#define toupper(c) ((c)>='a' && (c)<='z'? (c)+('A'-'a'): (c))
```

This is indeed faster than the function in many cases. However, it will cause a surprise for anyone who tries to use `toupper(*p++)`.

Another thing to watch out for when using macros is that they may generate very large expressions indeed. For example, look again at the definition of `max`:

```
#define max(a,b) ((a)>(b)?(a):(b))
```

Suppose we want to use this definition to find the largest of a, b, c, and d. If we write the obvious:

```
max(a,max(b,max(c,d)))
```

this expands to:

```
((a)>(((b)>(((c)>(d)?(c):(d))?(b):(((c)>(d)?(c):(d)))))?(a):(((b)>((c)>(d)?(c):(d))?(b):(((c)>(d)?(c):(d))))))
```

which is surprisingly large. We can make it a little less large by balancing the operands:

```
max(max(a,b),max(c,d))
```

which gives:

```
((((a)>(b)?(a):(b))>(((c)>(d)?(c):(d))?(((a)>(b)?(a):(b)):(((c)>(d)?(c):(d))))))
```

Somehow, though, it seems easier to write:

```
biggest = a;
if (biggest < b) biggest = b;
if (biggest < c) biggest = c;
if (biggest < d) biggest = d;
```

6.2. Macros are not Type Definitions

One common use of macros is to permit several things in diverse places to be the same type:

```
#define FOOTYPE struct foo
FOOTYPE a;
FOOTYPE b, c;
```

This lets the programmer change the types of `a`, `b`, and `c` just by changing one line of the program, even if `a`, `b`, and `c` are declared in widely different places.

Using a macro definition for this has the advantage of portability – any C compiler supports it. Most C compilers also support another way of doing this:

```
typedef struct foo FOOTYPE;
```

This defines `FOOTYPE` as a new type that is equivalent to `struct foo`.

These two ways of naming a type may appear to be equivalent, but the `typedef` is more flexible. Consider, for example, the following:

```
#define T1 struct foo *
typedef struct foo *T2;
```

These definitions make `T1` and `T2` conceptually equivalent to a pointer to a `struct foo`. But look what happens when we try to use them with more than one variable:

```
T1 a, b;
T2 c, d;
```

The first declaration gets expanded to

```
struct foo * a, b;
```

This defines `a` to be a pointer to a structure, but defines `b` to be a structure (not a pointer). The second declaration, in contrast, defines both `c` and `d` as pointers to structures, because `T2` behaves as a true type.

7. Portability Pitfalls

C has been implemented by many people to run on many machines. Indeed, one of the reasons to write programs in C in the first place is that it is easy to move them from one programming environment to another.

However, because there are so many implementors, they do not all talk to each other. Moreover, different systems have different requirements, so it is reasonable to expect C implementations to differ slightly between one machine and another.

Because so many of the early C implementations were associated with the UNIX operating system, the nature of many of these functions was shaped by that system. When people started implementing C under other systems, they tried to make the library behave in ways that would be familiar to programmers used to the UNIX system.

They did not always succeed. What is more, as more people in different parts of the world started working on different versions of the UNIX system, the exact nature of some of the library functions inevitably diverged. Today, a C programmer who wishes to write programs useful in someone else's environment must know about many of these subtle differences.

7.1. What's in a Name?

Some C compilers treat all the characters of an identifier as being significant. Others ignore characters past some limit when storing identifiers. C compilers usually produce object programs that must then be processed by loaders in order to be able to access library subroutines. Loaders, in turn, often impose their own restrictions on the kinds of names they can handle.

One common loader restriction is that letters in external names must be in upper case only. When faced with such a restriction, it is reasonable for a C implementor to force all external names to upper case. Restrictions of this sort are blessed by section 2.1 the C reference manual:

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _

counts as as a letter. Upper and lower case letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

Here, the reference manual goes on to give examples of various implementations that restrict external identifiers to a single case, or to fewer than eight characters, or both.

Because of all this, it is important to be careful when choosing identifiers in programs intended to be portable. Having two subroutines named, say `print_fields` and `print_float` would not be a very good idea.

As a striking example, consider the following function:

```
char *
Malloc (n)
    unsigned n;
{
    char *p, *malloc();
    p = malloc (n);
    if (p == NULL)
        panic ("out of memory");
    return p;
}
```

This function is a simple way of ensuring that running out of memory will not go undetected. The idea is for the program to allocate memory by calling `Malloc` instead of `malloc`. If `malloc` ever fails, the result will be to call `panic` which will presumably terminate the program with an appropriate error message.

Consider, however, what happens when this function is used on a system that ignores case distinctions in external identifiers. In effect, the names `malloc` and `Malloc` become equivalent. In other words, the library function `malloc` is effectively replaced by the `Malloc` function above, which when it calls `malloc` is really calling itself. The result, of course, is that the first attempt to allocate memory results in a recursion loop and consequent mayhem, even though the function will work on an implementation that preserves case distinctions.

7.2. How Big is an Integer?

C provides the programmer with three sizes of integers: ordinary, short, and long, and with characters, which behave as if they were small integers. The language definition does not guarantee much about the relative sizes of the various kinds of integer:

1. The four sizes of integers are non-decreasing.
2. An ordinary integer is large enough to contain any array subscript.
3. The size of a character is natural for the particular hardware.

Most modern machines have 8-bit characters, though a few have 7- or 9-bit characters, so characters are usually 7, 8, or 9 bits.

Long integers are usually at least 32 bits long, so that a long integer can be used to represent the size of a file.

Ordinary integers are usually at least 16 bits long, because shorter integers would impose too much of a restriction on the maximum size of an array.

Short integers are almost always exactly 16 bits long.

What does this all mean in practice? The most important thing is that one cannot count on having any particular precision available. Informally, one can probably expect 16 bits for a short or an ordinary integer, and 32 bits for a long integer, but not even those sizes are guaranteed. One can certainly use ordinary integers to express table sizes and subscripts, but what about a variable that must be able to hold values up to ten million?

The most portable way to do that is probably to define a “new” type:

```
typedef long tenmil;
```

Now one can use this type to declare a variable of that width and know that, at worst, one will have to change a single type definition to get all those variables to be the right type.

7.3. Are Characters Signed or Unsigned?

Most modern computers support 8-bit characters, so most modern C compilers implement characters as 8-bit integers. However, not all compilers interpret those 8-bit quantities the same way.

The issue becomes important only when converting a `char` quantity to a larger integer. Going the other way, the results are well-defined: excess bits are simply discarded. But a compiler converting a `char` to an `int` has a choice: should it treat the `char` as a signed or an unsigned quantity? If the former, it should expand the `char` to an `int` by replicating the sign bit; if the latter, it should fill the extra bit positions with zeroes.

The results of this decision are important to virtually anyone who deals with characters with their high-order bits turned on. It determines whether 8-bit characters are going to be considered to range from -128 through 127 or from 0 through 255 . This, in turn, affects the way the programmer will design things like hash tables and translate tables.

If you care whether a character value with the high-order bit on is treated as a negative number, you should probably declare it as `unsigned char`. Such values are guaranteed to be zero-extended when converted to integer, whereas ordinary `char` variables may be signed in one implementation and `unsigned` in another.

Incidentally, it is a common misconception that if `c` is a character variable, one can obtain the `unsigned` integer equivalent of `c` by writing `(unsigned) c`. This fails because a `char` quantity is converted to `int` before any operator is applied to it, *even a cast*. Thus `c` is converted first to a signed integer and then to an `unsigned` integer, with possibly unexpected results.

The right way to do it is `(unsigned char) c`.

7.4. Are Right Shifts Signed or Unsigned?

This bears repeating: a program that cares how shifts are done had better declare the quantities being shifted as `unsigned`.

7.5. How Does Division Truncate?

Suppose we divide `a` by `b` to give a quotient `q` and remainder `r`:

```
q = a / b;  
r = a % b;
```

For the moment, suppose also that $b > 0$.

What relationships might we want to hold between `a`, `b`, `p`, and `q`?

1. Most important, we want $q * b + r == a$, because this is the relation that defines the remainder.
2. If we change the sign of `a`, we want that to change the sign of `q`, but not the absolute value.
3. We want to ensure that $r \geq 0$ and $r < b$. For instance, if the remainder is being used as an index to a hash table, it is important to be able to know that it will always be a valid index.

These three properties are clearly desirable for integer division and remainder operations. Unfortunately, *they cannot all be true at once*.

Consider $3/2$, giving a quotient of 1 and a remainder of 1 . This satisfies property 1. What should be the value of $-3/2$? Property 2 suggests that it should be -1 , but if that is so, the remainder must *also* be -1 , which violates property 3. Alternatively, we can satisfy property 3 by making the remainder 1 , in which case property 1 demands that the quotient be -2 . This violates property 2.

Thus C, and any language that implements truncating integer division, must give up at least one of

these three principles.

Most programming languages give up number 3, saying instead that the remainder has the same sign as the dividend. This makes it possible to preserve properties 1 and 2. Most C implementations do this in practice, also.

However, the C language definition only guarantees property 1, along with the property that $|r| < |b|$ and that $r \geq 0$ whenever $a \geq 0$ and $b > 0$. This property is less restrictive than either property 2 or property 3, and actually permits some rather strange implementations that would be unlikely to occur in practice (such as an implementation that always truncates the quotient *away from zero*).

Despite its sometimes unwanted flexibility, the C definition is enough that we can usually make integer division do what we want, provided that we know what we want. Suppose, for example, that we have a number *n* that represents some function of the characters in an identifier, and we want to use division to obtain a hash table entry *h* such that $0 \leq h < \text{HASHSIZE}$. If we know that *n* is never negative, we simply write

```
h = n % HASHSIZE;
```

However, if *n* might be negative, this is not good enough, because *h* might also be negative. However, we know that $h > -\text{HASHSIZE}$, so we can write:

```
h = n % HASHSIZE;
if (h < 0)
    h += HASHSIZE;
```

Better yet, declare *n* as *unsigned*.

7.6. How Big is a Random Number?

This size ambiguity has affected library design as well. When the only C implementation ran on the PDP-11‡ computer, there was a function called *rand* that returned a (pseudo-) random non-negative integer. PDP-11 integers were 16 bits long, including the sign, so *rand* would return an integer between 0 and $2^{15} - 1$.

When C was implemented on the VAX-11, integers were 32 bits long. What was the range of the *rand* function on the VAX-11?

For their system, the people at the University of California took the view that *rand* should return a value that ranges over all possible non-negative integers, so their version of *rand* returns an integer between 0 and $2^{31} - 1$.

The people at AT&T, on the other hand, decided that a PDP-11 program that expected the result of *rand* to be less than 2^{15} would be easier to transport to a VAX-11 if the *rand* function returned a value between 0 and 2^{15} there, too.

As a result, it is now difficult to write a program that uses *rand* without tailoring it to the implementation.

7.7. Case Conversion

The *toupper* and *tolower* functions have a similar history. They were originally written as macros:

```
#define toupper(c) ((c) + 'A' - 'a')
#define tolower(c) ((c) + 'a' - 'A')
```

When given a lower-case letter as input *toupper* yields the corresponding upper-case letter. *Tolower* does the opposite. Both these macros depend on the implementation's character set to the extent that they demand that the difference between an upper-case letter and the corresponding lower-case letter be the same constant for all letters. This assumption is valid for both the ASCII and EBCDIC character sets, and probably isn't too dangerous, because the non-portability of these macro definitions can be encapsulated in

‡ PDP-11 and VAX-11 are Trademarks of Digital Equipment Corporation.

the single file that contains them.

These macros do have one disadvantage, though: when given something that is not a letter of the appropriate case, they return garbage. Thus, the following innocent program fragment to convert a file to lower case doesn't work with these macros:

```
int c;
while ((c = getchar()) != EOF)
    putchar (tolower (c));
```

Instead, one must write:

```
int c;
while ((c = getchar()) != EOF)
    putchar (isupper (c)? tolower (c): c);
```

At one point, some enterprising soul in the UNIX development organization at AT&T noticed that most uses of *toupper* and *tolower* were preceded by tests to ensure that their arguments were appropriate. He considered rewriting the macros this way:

```
#define toupper(c) ((c) >= 'a' && (c) <= 'z'? (c) + 'A' - 'a': (c))
#define tolower(c) ((c) >= 'A' && (c) <= 'Z'? (c) + 'a' - 'A': (c))
```

but realized that this would cause *c* to be evaluated anywhere between one and three times for each call, which would play havoc with expressions like *toupper*(**p*++). Instead, he decided to rewrite *toupper* and *tolower* as functions. *Toupper* now looked something like this:

```
int toupper (c)
    int c;
{
    if (c >= 'a' && c <= 'z')
        return c + 'A' - 'a';
    return c;
}
```

and *tolower* looked similar.

This change had the advantage of robustness, at the cost of introducing function call overhead into each use of these functions. Our hero realized that some people might not be willing to pay the cost of this overhead, so he re-introduced the macros with new names:

```
#define _toupper(c) ((c)+'A'-'a')
#define _tolower(c) ((c)+'a'-'A')
```

This gave users a choice of convenience or speed.

There was just one problem in all this: the people at Berkeley never followed suit, nor did some other C implementors. This means that a program written on an AT&T system that uses *toupper* or *tolower*, and assumes that it will be able to pass an argument that is not a letter of the appropriate case, may stop working on some other C implementation.

This sort of failure is very hard to trace for someone who does not know this bit of history.

7.8. Free First, then Reallocate

Most C implementations provide users with three memory allocation functions called *malloc*, *realloc*, and *free*. Calling *malloc(n)* returns a pointer to *n* characters of newly-allocated memory that the programmer can use. Giving *free* a pointer to memory previously returned by *malloc* makes that memory available for re-use. Calling *realloc* with a pointer to an allocated area and a new size stretches or shrinks the memory to the new size, possibly copying it in the process.

Or so one might think. The truth is actually somewhat more subtle. Here is an excerpt from the description of *realloc* that appears in the System V Interface Definition:

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

The Seventh Edition of the reference manual for the UNIX system contains a copy of the same paragraph. In addition, it contains a second paragraph describing *realloc*:

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Thus, the following is legal under the Seventh Edition:

```
free (p);
p = realloc (p, newsiz);
```

This idiosyncrasy remains in systems derived from the Seventh Edition: it is possible to free a storage area and then reallocate it. By implication, freeing memory on these systems is guaranteed not to change its contents until the next time memory is allocated. Thus, on these systems, one can free all the elements of a list by the following curious means:

```
for (p = head; p != NULL; p = p->next)
    free ((char *) p);
```

without worrying that the call to *free* might invalidate *p->next*.

Needless to say, this technique is not recommended, if only because not all C implementations preserve memory long enough after it has been freed. However, the Seventh Edition manual leaves one thing unstated: the original implementation of *realloc* actually required that the area given to it for reallocation be free first. For this reason, there are many C programs floating around that free memory first and then reallocate it, and this is something to watch out for when moving a C program to another implementation.

7.9. An Example of Portability Problems

Let's take a look at a problem that has been solved many times by many people. The following program takes two arguments: a long integer and a (pointer to a) function. It converts the integer to decimal and calls the given function with each character of the decimal representation.

```
void
printnum (n, p)
    long n;
    void (*p)();
{
    if (n < 0) {
        (*p) ('-');
        n = -n;
    }
    if (n >= 10)
        printnum (n/10, p);
    (*p) (n % 10 + '0');
}
```

This program is fairly straightforward. First we check if *n* is negative; if so, we print a sign and make *n* positive. Next, we test if $n \geq 10$. If so, its decimal representation has two or more digits, so we call *printnum* recursively to print all but the last digit. Finally, we print the last digit.

This program, for all its simplicity, has several portability problems. The first is the method it uses to convert the low-order decimal digit of *n* to character form. Using $n \% 10$ to get the value of the low-order digit is fine, but adding '0' to it to get the corresponding character representation is not. This addition assumes that the machine collating sequence has all the digits in sequence with no gaps, so that '0' + 5 has the same value as '5', and so on. This assumption, while true of the ASCII and EBCDIC character sets, might not be true for some machines. The way to avoid that problem is to use a table:

```

void
printnum (n, p)
    long n;
    void (*p)();
{
    if (n < 0) {
        (*p) ('-');
        n = -n;
    }
    if (n >= 10)
        printnum (n/10, p);
    (*p) ("0123456789"[n % 10]);
}

```

The next problem involves what happens if $n < 0$. The program prints a negative sign and sets n to $-n$. This assignment might overflow, because 2's complement machines generally allow more negative values than positive values to be represented. In particular, if a (long) integer is k bits plus one extra bit for the sign, -2^k can be represented but 2^k cannot.

There are several ways around this problem. The most obvious one is to assign n to an `unsigned long` value and be done with it. However, some C compilers do not implement `unsigned long`, so let us see how we can get along without it.

In both 1's complement and 2's complement machines, changing the sign of a *positive* integer is guaranteed not to overflow. The only trouble comes when changing the sign of a *negative* value. Therefore, we can avoid trouble by making sure we do not attempt to make n positive.

Of course, once we have printed the sign of a negative value, we would like to be able to treat negative and positive numbers the same way. The way to do that is to force n to be negative after printing the sign, and to do all our arithmetic with negative values. If we do this, we will have to ensure that the part of the program that prints the sign is executed only once; the easiest way to do that is to split the program into two functions:

```

void
printnum (n, p)
    long n;
    void (*p)();
{
    void printneg();
    if (n < 0) {
        (*p) ('-');
        printneg (n, p);
    } else
        printneg (-n, p);
}

void
printneg (n, p)
    long n;
    void (*p)();
{
    if (n <= -10)
        printneg (n/10, p);
    (*p) ("0123456789"[-(n % 10)]);
}

```

Printnum now just checks if the number being printed is negative; if so it prints a negative sign. In

either case, it calls *printneg* with the negative absolute value of *n*. We have also modified the body of *printneg* to cater to the fact that *n* will always be a negative number or zero.

Or have we? We have used *n/10* and *n%10* to represent the leading digits and the trailing digit of *n* (with suitable sign changes). Recall that integer division behaves in a somewhat implementation-dependent way when one of the operands is negative. For that reason, it might actually be that *n%10* is positive! In that case, $-(n \% 10)$ would be negative, and we would run off the end of our digit array.

We cater to this problem by creating two temporary variables to hold the quotient and remainder. After we do the division, we check that the remainder is in range and adjust both variables if not. *Printnum* has not changed, so we show only *printneg*:

```
void
printneg (n, p)
    long n;
    void (*p)();
{
    long q;
    int r;

    q = n / 10;
    r = n % 10;
    if (r > 0) {
        r -= 10;
        q++;
    }
    if (n <= -10)
        printneg (q, p);
    (*p) ("0123456789"[-r]);
}
```

8. This Space Available

There are many ways for C programmers to go astray that have not been mentioned in this paper. If you find one, please contact the author. It may well be included, with an acknowledging footnote, in a future revision.

References

The C Programming Language (Kernighan and Ritchie, Prentice-Hall 1978) is the definitive work on C. It contains both an excellent tutorial, aimed at people who are already familiar with other high-level languages, and a reference manual that describes the entire language succinctly. While the language has expanded slightly since 1978, this book is still the last word on most subjects. This book also contains the ‘‘C Reference Manual’’ we have mentioned several times in this paper.

The C Puzzle Book (Feuer, Prentice-Hall, 1982) is an unusual way to hone one’s syntactic skills. The book is a collection of puzzles (and answers) whose solutions test the reader’s knowledge of C’s fine points.

C: A Reference Manual (Harbison and Steele, Prentice Hall 1984) is mostly intended as a reference source for implementors. Other users may also find it useful, particularly because of its meticulous cross references.

Bijlage D

C++ Tips and Traps

Dit document geeft een overzicht aan C programmeurs met praktische tips en veelgemaakte fouten in C++. Volgende bladzijden in deze bijlage zijn interessant om door te nemen:

- blz 1 t.e.m. blz 13
- blz 20 t.e.m. blz 28

The C++ Programming Language

C++ Tips and Traps

Outline

Tips for C Programmers
C++ Traps and Pitfalls
Efficiency and Performance

1

Tips for C Programmers

- Use `const` instead of `#define` to declare program constants, e.g.,

– C

```
#define PI 3.14159
#define MAX_INT 0x7FFFFFFF
#define MAX_UNSIGNED 0xFFFFFFFF
```

– C++

```
const double PI = 3.14159;
const int MAX_INT = 0x7FFFFFFF;
const unsigned MAX_UNSIGNED = 0xFFFFFFFF;
```

- Names declared with `#define` are untyped and unrestricted in scope

– In contrast, names declared with `const` are typed and follow C++ scope rules

* e.g., `consts` have static linkage...

2

Tips for C Programmers (cont'd)

- Use inline functions and parameterized types instead of preprocessor macros, e.g.,

– C

* Macros

```
#define MAX(A,B) (((A) >= (B)) ? (A) : (B))
/* ... */
MAX (a++, b++); /* Trouble! */
```

* Using a type as a parameter:

```
#define DECLARE_MAX(TYPE) \
    TYPE MAX (TYPE a, TYPE b) \
    { return a >= b ? a : b; }
DECLARE_MAX (int)
DECLARE_MAX (double)
DECLARE_MAX (char)
```

– C++

```
inline int MAX (int a, int b) {return a >= b ? a : b;}
/* ... */
MAX (a++, b++); /* No problem! */
template <class T> inline
MAX (T a, T b) { return a >= b ? a : b; }
```

3

Tips for C Programmers (cont'd)

- Note, there are still some uses for preprocessor, however, e.g.,

– Wrapping headers and commenting out code blocks:

```
#ifndef _FOOBAR_H
#define _FOOBAR_H
...
#endif
```

– Stringizing and token pasting

```
#define name2(A,B) A##B
```

– File inclusion

```
#include <iostream.h>
```

4

Tips for C Programmers (cont'd)

- Be careful to distinguish between **int** and **unsigned**
- Unlike C, C++ distinguishes between **int** and **unsigned int**, so be careful when using overloaded functions:

```
#include <iostream.h>
inline void f (int) { cout << "f (int) called\n"; }
inline void f (unsigned) { cout << "f (unsigned) called\n"; }
int main (void) {
    f (1); // calls f (int)
    f (1U); // calls f (unsigned)
}
```

5

Tips for C Programmers (cont'd)

- Consider using references instead of pointers as function arguments, e.g.,

– C

```
void screen_size (unsigned *height, unsigned *width);
/* ... */
unsigned height, width;
screen_size (&height, &width);
```

– C++

```
void screen_size (unsigned &height, unsigned &width);
// ...
unsigned height, width;
screen_size (height, width);
```

- However, it is harder to tell if arguments are modified with this approach!

6

Tips for C Programmers (cont'd)

- Declare reference or pointer arguments that are not modified by a function as **const**, e.g.,

– C

```
struct Big_Struct { int array[100000], int size; };

void foo (struct Big_Struct *bs);
// passed as pointer for efficiency

int strlen (char *str);
```

– C++

```
void foo (const Big_Struct &bs);

int strlen (const char *str);
```

- This allows callers to use **const** values as arguments and also prevents functions from accidentally modifying their arguments

7

Tips for C Programmers (cont'd)

- Use overloaded function names instead of different function names to distinguish between functions that perform the same operations on different data types:

– C

```
int abs (int x);
double fabs (double x);
long labs (long x);
```

– C++

```
int abs (int x);
double abs (double x);
long abs (long x);
```

- Do not forget that C++ does *NOT* permit overloading on the basis of return type!

8

Tips for C Programmers (cont'd)

- Use **new** and **delete** instead of **malloc** and **free**, e.g.,

– C

```
int size = 100;
int *ipa = malloc (size); /* Error!!! */
/* ...*/
free (ipa);
```

– C++

```
const int size = 100;
int *ipa = new int[size];
// ...
delete [] ipa;
```

- **new** can both help avoid common errors with **malloc** and also ensure that constructors and destructors are called

9

Tips for C Programmers (cont'd)

- Use iostream I/O operators **<<** and **>>** instead of **printf** and **scanf**

– C

```
float x;
scanf ("%f", &x);
printf ("The answer is %f\n", x);
fprintf (stderr, "Invalid command\n");
```

– C++

```
cin >> x;
cout << "The answer is " << x << "\n";
cerr << "Invalid command\n";
```

- The **<<** and **>>** stream I/O operators are
 - (1) type-safe and (2) extensible to user-defined types

10

Tips for C Programmers (cont'd)

- Use **static** objects with constructor/destructors instead of explicitly calling initialization/finalization functions

– C

```
struct Symbol_Table {
    /* ...*/
};

void init_symbol_table (struct Symbol_Table *);
int lookup (struct Symbol_Table *);
static struct Symbol_Table sym_tab;
int main (void) {
    char s[100];
    init_symbol_table (&sym_tab);
    /* ...*/
}
```

– C++

```
class Symbol_Table : private Hash_Table {
public:
    Symbol_Table (void); // init table
    int lookup (String &key);
    ~Symbol_Table (void);
};

static Symbol_Table sym_tab;
int main (void) {
    String s;
    while (cin >> s)
        if (sym_tab.lookup (s) != 0)
            cout << "found " << s << "\n";
}
```

11

Tips for C Programmers (cont'd)

- Declare variables near the place where they are used, and initialize variables in their declarations, e.g.,

– C

```
void dup_assign (char **dst, char *src) {
    int len;
    int i;
    if (src == *dst) return;
    if (*dst != 0) free (*dst);
    len = strlen (src);
    *dst = (char *) malloc (len + 1);
    for (i = 0; i < len; i++) (*dst)[i] = src[i];
}
```

– C++

```
void dup_assign (char *&dst, const char *src) {
    if (src == dst) return;
    delete dst; // delete checks for dst == 0
    int len = strlen (src);
    dst = new char[len + 1];
    for (int i = 0; i < len; i++) dst[i] = src[i];
}
```

12

Tips for C Programmers (cont'd)

- Use derived classes with virtual functions rather than using **switch** statements on type members:

– C

```
#include <math.h>
enum Shape_Type {
    TRIANGLE, RECTANGLE, CIRCLE
};

struct Triangle { float x1, y1, x2, y2, x3, y3; };
struct Rectange { float x1, y1, x2, y2; };
struct Circle { float x, y, r; };

struct Shape {
    enum Shape_Type shape;
    union {
        struct Triange t;
        struct Rectange r;
        struct Circle c;
    } u;
};
```

13

- C (cont'd)

```
float area (struct Shape *s) {
    switch (s->shape) {
        case TRIANGLE:
            struct Triangle *p = &s->u.t;
            return fabs (
                (p->x1 * p->y2 - p->x2 * p->y1) +
                (p->x2 * p->y3 - p->x3 * p->y2) +
                (p->x3 * p->y1 - p->x1 * p->y3)) / 2;
        case RECTANGLE:
            struct Rectange *p = &s->u.r;
            return fabs ((p->x1 - p->x2) *
                         (p->y1 - p->y2));
        case CIRCLE:
            struct Circle *p = &s->u.c;
            return M_PI * p->r * p->r;
        default:
            fprintf (stderr, "Invalid shape\n");
            exit (1);
    }
}
```

14

• C++

```
#include <iostream.h>
#include <math.h>
class Shape {
public:
    Shape () {}
    virtual float area (void) const = 0;
};
class Triangle : public Shape {
public:
    Triangle (float x1, float x2, float x3,
              float y1, float y2, float y3);
    virtual float area (void) const;
private:
    float x1, y1, x2, y2, x3, y3;
};
float Triangle::area (void) const {
    return fabs ((x1 * y2 - x2 * y1) +
                 (x2 * y3 - x3 * y2) +
                 (x3 * y1 - x1 * y3)) / 2;
}
```

15

• C++

```
class Rectange : public Shape {
public:
    Rectangle (float x1, float y1, float x2, float y2);
    virtual float area (void) const;
private:
    float x1, y1, x2, y2;
};
float Rectangle::area (void) const {
    return fabs ((x1 - x2) * (y1 - y2));
}
class Circle : public Shape {
public:
    Circle (float x, float y, float r);
    virtual float area (void) const;
private:
    float x, y, r;
};
float Circle::area (void) const {
    return M_PI * r * r;
}
```

16

Tips for C Programmers (cont'd)

- Use static member variables and functions instead of global variables and functions, and place **enum** types in class declarations
- This approach avoid polluting the global name space with identifiers, making name conflicts less likely for libraries
 - C

```
#include <stdio.h>
enum Color_Type { RED, GREEN, BLUE };
enum Color_Type color = RED;
unsigned char even_parity (void);
int main (void) {
    color = GREEN;
    printf ("% .2x\n", even_parity ('Z'));
}
```

17

Tips for C Programmers (cont'd)

- static members (cont'd)

- C++

```
#include <iostream.h>
class My_Lib {
public:
    enum Color_Type { RED, GREEN, BLUE };
    static Color_Type color;
    static unsigned char even_parity (char c);
};
My_Lib::Color_Type My_Lib::color = My_Lib::RED;
int main (void) {
    My_Lib::color = My_Lib::GREEN;
    cout << hex (int (My_Lib::even_parity ('Z')))
        << "\n";
}
```

- Note that the new C++ "namespaces" feature will help solve this problem even more elegantly

18

Tips for C Programmers (cont'd)

- Use anonymous unions to eliminate unnecessary identifiers

- C

```
unsigned hash (double val) {
    static union {
        unsigned asint[2];
        double asdouble;
    } u;
    u.asdouble = val;
    return u.asint[0] ^ u.asint[1];
}
```

- C++

```
unsigned hash (double val) {
    static union {
        unsigned asint[2];
        double asdouble;
    };
    asdouble = val;
    return asint[0] ^ asint[1];
}
```

19

C++ Traps and Pitfalls

- Ways to circumvent C++'s protection scheme:

```
#define private public
#define const
#define class struct
```

- Note, in the absence of exception handling it is very difficult to deal with constructor failures

- e.g., in operator overloaded expressions that create temporaries

20

C++ Traps and Pitfalls (cont'd)

- Initialization vs Assignment

- Consider the following code

```

class String {
public:
    String (void); // Make a zero-len String
    String (const char *s); // char * --> String
    String (const String &s); // copy constructor
    String &operator= (const String &s); // assignment
private:
    int len;
    char *data;
};

class Name {
public:
    Name (const char *t) { s = t; }
private:
    String s;
};

int main (void) {
    // How expensive is this?????????
    Name neighbor = "Joe";
}

```

21

- Initialization vs Assignment (cont'd)

- Constructing “neighbor” object is costly

1. `Name::Name` gets called with parameter “Joe”
2. `Name::Name` has no base initialization list, so member object “`neighbor.s`” is constructed by default `String::String`
 - * This will probably allocate a 1 byte area from freestore for the ‘\0’
3. A temporary “Joe” String is created from parameter `t` using the `CONST CHAR *` constructor
 - * This is another freestore allocation and a `strcpy`
4. `String::operator= (const string &)` is called with the temporary String
5. This will **delete** the old string in `s`, use another **new** to get space for the new string, and do another `strcpy`

22

- 6. The temporary String gets destroyed, yet another freestore operation

- Final score: 3 **new**, 2 **strcpy**, and 2 **delete**
Total “cost units”: 7

- Initialization vs Assignment (cont'd)

- Compare this to an initialization-list version. Simply replace

```

Name::Name (const char* t) { s = t; }
with
Name::Name (const char* t): s (t) { }

```

- Now construction of “neighbor” is:

1. `Name::Name (const char *)` gets called with parameter “Joe”
2. `Name::Name (const char *)` has an init list, so `neighbor::s` is initialized from `S` with `String::String (const char *)`
3. `String::String ("Joe")` will probably do a **new** and a `strcpy`

- Final score: 1 **new**, 1 **strcpy**, and 0 **delete**
Total “cost units”: 2

- Conclusion: *always* use the initialization syntax, even when it does not matter...

23

C++ Traps and Pitfalls (cont'd)

- Although a function with no arguments must be called with empty parens a constructor with no arguments must be called with *no* parens!

```
class Foo {
public:
    Foo (void);
    int bar (void);
};

int main (void) {
    Foo f;
    Foo ff (); // declares a function returning Foo!
    f.bar (); // call method
    f.bar; // a no-op
    ff.bar (); // error!
}
```

24

- Default Parameters and Virtual Functions

```
extern "C" int printf (const char *, ...);

class Base {
public:
    virtual void f (char *name = "Base") {
        printf ("base = %s\n", name);
    }
};

class Derived : public Base {
public:
    virtual void f (char *name = "Derived") {
        printf ("derived = %s\n", name);
    }
};

int main (void) {
    Derived *dp = new Derived;
    dp->f (); /* prints "derived = Derived" */

    Base *bp = dp;
    bp->f (); /* prints "derived = Base" */
    return 0;
}
```

25

C++ Traps and Pitfalls (cont'd)

- Beware of subtle whitespace issues...

```
int b = a /* divided by 4 *//4;
-a;
/* C++ preprocessing and parsing */
int b = a -a;
/* C preprocessing and parsing */
int b = a/4; -a;
```

- Note, in general it is best to use whitespace around operators and other syntactic elements, e.g.,

```
char *x;
int foo (char * = x); // OK
int bar (char*=x); // Error
```

26

Efficiency and Performance

- *Inline Functions*

- Use of inlines in small programs can help performance, extensive use of inlines in large projects can actually hurt performance by enlarging code, bringing on paging problems, and forcing many recompilations
- Sometimes it's good practice to turn-off inlining to set a worst case performance base for your application, then go back an inline as part of performance tuning

- *Parameter Passing*

- Passing C++ objects by reference instead of value is a good practice
 - * It's rarely to your advantage to replicate data and fire off constructors and destructors unnecessarily

27

Efficiency and Performance (cont'd)

- *Miscellaneous Tips*
 - Use good memory (heap) management strategies
 - Develop good utility classes (for strings, in particular)
 - Good object and protocol design (particularly, really isolating large-grained objects)
 - Give attention to paging and other ways your application uses system resources
- While C++ features, if used unwisely, can slow an application down, C++ is not inherently slower than say C, particularly for large scale projects
 - In fact, as the size and complexity of software increases, such comparisons aren't even relevant since C fails to be a practical approach whereas C++ comes into its own