Informatik I – Grundlagen, Algorithmen-Entwurf, C/C++

Dr. Stefan Lewandowski*

Version 7. Januar 2011

Dieses Skript darf nur mit ausdrücklicher Zustimmung des Autors vervielfältigt werden. Die Weiterverwendung des Skripts (auch in Auszügen) und der zugehörigen Programme bedarf der Zustimmung des Autors.

Inhaltsverzeichnis

0 Einführung **4**

0.1 Ein kurzer Überblick 4

0.1.1 Definition Informatik 4

0.1.2 Inhaltliche Ziele der Veranstaltung „Informatik I und II“ 5

0.1.3 Umfang und Anforderungen (Anwesenheit reicht nicht!) 6

0.1.4 Voraussetzungen (Wissen / Können aus der Zeit vor dem Studium) . . . 6

0.1.5 Lernziele 7

0.1.6 Materialien und Literatur 7

0.1.7 Allgemeine Hinweise zum Programmieren 8

0.1.8 Der Softwareentwicklungsprozess – ein Beispielproblem 8

1 Programmierung – eine Einführung in den strukturierten Entwurf und dessen Implementierung in C/C++ **11**

1.1 Pseudo-Code und Nassi-Shneiderman-Diagramme 11

1.1.1 Das Spiel Mensch-Ärgere-Dich-Nicht 11

1.1.2 Übungsbeispiel: Winterreifenwechsel 14

1.2 Definition Algorithmus und Programm 14

1.2.1 Ein Algorithmus ist 14

1.2.2 Was unterscheidet ein Programm von einem Algorithmus? 15

1.2.3 Übungsbeispiel: Ist das ein Algorithmus? 16

1.3 Die Rollenspiel-Metapher 16

1.3.1 Programmierer, Ausführer und Benutzer 16

*E-Mail: slewand@slewand.de

1.3.2 Compiler und Interpreter 18
1.4 Sprachen zur Beschreibung der Syntax von Sprachen 18
1.4.1 EBNF 20
1.4.2 Syntaxdiagramme 20
1.5 Variablen und Datentypen 21
1.6 Vom Problem zum Programm 23
1.6.1 Erweiterte Struktogramme, das EVA-Prinzip, Datenflussdiagramme (DF) und Programmablaufpläne (PA) 24
1.6.2 Strukturierter Entwurf und dessen Umsetzung in C/C++ 26
1.6.3 Parameterübergabe-Mechanismen und ein einfacher Sortieralgorithmus . 31
1.6.4 Das Programmhierarchiediagramm (PH) 33
1.6.5 Lebensdauer, Gültigkeit und Sichtbarkeit von Variablen 34
1.6.6 Eigene Datentypen mit `typedef` und `struct` und das Datenhierarchiediagramm (DH) 34
1.6.7 Programmablaufplan mit Daten (PAD), Programmnetz (PN) und Datenflussdiagramme (DF) höherer Ebenen 41
1.6.8 Abstrakte Datentypen (ADT) und Modularisierung 42
1.6.9 Der Programmcode „Mensch-Ärgere-Dich-Nicht“ und Known Bugs . . . 45
1.7 Rekursion und das Rückgabekonzept von C/C++ 56
1.7.1 Einschub: das Rückgabekonzept von C/C++ 57
1.7.2 Die Türme von Hanoi 58
1.8 Zeiger (Pointer) und Abstrakte Datentypen (ADT) 59
1.8.1 Die ADTs Stack und Queue 60
1.8.2 Dynamische Variablen in C/C++ mit `new` und `delete` (oder `malloc` bzw. `free`) 62
1.8.3 Der ADT Liste 65
1.8.4 Varianten von Listen oder der ADT Queue 66
1.8.5 C/C++-Code der Module stack, queue und liste 67

2 Endliche Automaten **73**

A Kompilieren der Übungsbeispiele **73**
A.1 Microsoft Visual Studio 73
A.2 MS Visual Studio C++ 2005 / 2008 Express 74
A.3 Bloodshed Dev-C++ 75
A.4 Kompilieren von der Kommandozeile 76
A.4.1 GNU gcc 76
A.4.2 MinGW 76

B Begleitmaterialien und Aufgaben für das Labor **77**

B.1 Ein erstes C/C++-Programm 77

B.2 Die `if`-Anweisung 77

B.3 Allgemeine Hinweise zur Lösung der Übungsaufgaben 78

B.4 Einfache Zählschleifen 79

B.5 Unterprogramme 79

B.6 Kopf- und fußgesteuerte Schleifen – und – Zahlendarstellung im Rechner 81

B.7 Eigene Datentypen mit `typedef`, `struct` und `enum` 82

B.8 Programmhierarchie, Erstellung und Test von umfangreichen Programmen . . . 84

B.9 Umfangreiche Übungsaufgabe: Lösung von Umfüllproblemen 85

B.10 Modularisierung 88

B.11 Pointer und Rekursion – Stacks, Listen, ... 93

C Zusätzliche Übungsaufgaben und alte Klausuraufgaben **98**

0 Einführung

Vorlesung und Labor

- 1. Semester: Informatik I : 3V + 4L
- Teilklausur Informatik I
- 2. Semester: Informatik II : 2V + 3L
- Teilklausur Informatik II
- ggf. Wiederholungsklausur Informatik I+II

0.1 Ein kurzer Überblick

0.1.1 Definition Informatik

Definiert man Mathematik, so würde (hoffentlich) keiner auf die Idee kommen zu sagen: „Mathematik ist alles, was mit Taschenrechnern zu tun hat“. Als Informatiker wird man hingegen öfters mit folgender „Definition“ konfrontiert:

> **Definition:** Informatik ist alles, was mit Computern zu tun hat

Das ist **falsch!** – wir zitieren hier die kompakte Definition aus dem Duden Informatik

> Informatik ist die Wissenschaft von der **systematischen** Darstellung, Speicherung, Verarbeitung und Übertragung von **Informationen**, besonders der **automatischen** Verarbeitung mithilfe von Digitalrechnern (Computer).

Lernziele sind also insbesondere **nicht** der Beherrschung von Windows oder Linux und der darauf laufenden Programme.

Informatik befasst sich mit dem Rohstoff „Information“ und seiner Darstellung, Verarbeitung, Speicherung und Übertragung, vor allem aus technischer Sicht. In den Vordergrund rücken immer stärker die „Informationsprozesse“ und deren Planung, Gestaltung, Realisierung und Einbettung in größere Systeme. Da das menschliche Zusammenleben auf dem Austausch von Informationen beruht, hat die Informatik nachhaltigen Einfluss auf unsere Gesellschaft, auf das Arbeitsleben und auf das Privatleben jedes einzelnen. Stichwörter sind: Automatisierung geistiger Abläufe, Dienstleistungs- und Wissensgesellschaft, Beherrschbarkeit von Prozessen und Methoden zur Bearbeitung unüberschaubarer Vorgänge. Der zielorientierte Umgang mit der Informationsverarbeitung gehört heute zur erfolgreichen Ausübung sehr vieler Berufe. Zum einen sichert diese Tatsache den meisten Hochschulabsolvent(inn)en der Informatik ihren Arbeitsplatz, zum anderen droht hier eine Spaltung der Bevölkerung und des Arbeitsmarktes, bekannt unter dem Begriff „digital divide“, je nachdem, wie viel Befähigungen jemand besitzt, um die Informationstechnologie verstehen und für seine Arbeit einsetzen zu können. Der Begriff besagt bereits, dass die Objekte und Produkte der Informatik „digitalisiert“ (gut untereinander unterscheidbar, zeichenartig) sind. Sie sind zugleich abstrakt und nicht „anfassbar“ (d. h., für die menschlichen Sinne nicht unmittelbar erfahrbar). Informatikprodukte erkennt man nur an den Auswirkungen.

Informatik ist heute überwiegend eine Ingenieurwissenschaft. Sie umfasst jedoch

- Grundlagen,
- ingenieurmäßige Vorgehensweisen und
- experimentelle Anteile.

Im ersten Studienjahr werden die Grundlagen, insbesondere die Prinzipien und Konzepte, sowie mehrere ingenieurmäßige Vorgehensweisen, vor allem konstruktive Verfahren und ihre Umsetzung in eine konkrete Programmiersprache, betont.

Für das Verständnis der Informatik sind die Abstraktion und die Formalisierung besonders wichtig. Abstraktion bezeichnet das Herausarbeiten des Wesentlichen, wobei man sich von vielen konkreten Details befreit. Unter Abstraktion versteht man auch den Übergang zu einer Metaebene; sie umfasst das Erarbeiten gemeinsamer Ideen oder Prinzipien aus den unterschiedlichsten (Anwendungs-) Gebieten, z. B. die einheitliche Darstellung von Strukturen in Programmiersprachen, von syntaktischen Gebilden bei natürlichen Sprachen oder von Rechenbäumen mit Hilfe von (kontextfreien) Grammatiken. Abstraktes Denken ist eine wichtige Voraussetzung für eine adäquate Modellierung, wobei die Dinge der realen Welt in eine Modellierungssprache übertragen werden, aus der heraus man die Modelle dann weiter in Programmiersprachen abbildet. Um diese Modelle maschinell verarbeiten zu können, müssen sie formalisiert werden. Formalisierung beschreibt den Prozess, Geschehnisse der realen Welt oder (unscharfe) Informationen zu präzisieren und zugleich in Kalkülen zu notieren.

0.1.2 Inhaltliche Ziele der Veranstaltung „Informatik I und II“

Ziele der Veranstaltung sind die Vermittlung grundlegender Darstellungen und Prinzipien der Informatik, insbesondere

- Formalismen und Beschreibungssprachen, Modellbildung,
- Algorithmen und ihre Eigenschaften (Zeitkomplexität, Verifikation, Terminierung),
- Datenstrukturen und die Effizienz zugehöriger Verfahren,
- Grundbegriffe und Denkweisen der Programmierung,
- Abstraktion und Theorie, auf denen diese Begriffe basieren,
- Vorgehensweisen zur Problemlösung und die Realisierung und Implementierung der Konzepte,
- Beherrschung wichtiger Standardalgorithmen,
- genaue Kenntnis typischer Beispiele und ihrer Programmierung,
- genaue Kenntnis einer Programmiersprache (hier zunächst der klassische Kern von C und C++).

Die Vorlesung behandelt die Basisstrukturen und Konstruktionsprinzipien (heutiger) informationsverarbeitender Systeme, stellt wichtige Denkweisen zur Entwicklung von Verfahren vor, vermittelt langlebige Ideen und Methoden und diskutiert Beurteilungskriterien über Vor- und Nachteile von Prozessen und allgemein von Problemlösungen.

Die Vorlesung wird ergänzt durch das Labor, in dem die Programmiersprache C „wie ein Handwerk“ gelehrt wird. Hier werden die in der Grundvorlesung behandelten Konzepte an Beispielprogrammen eingeübt und die erarbeiteten Lösungen in Programme übertragen. Parallel dazu werden auch in der Vorlesung Übungsaufgaben gestellt und besprochen.

Zur Bedeutung der Programmiersprache: Bundesweit wurde in den letzten Jahren an den deutschen Hochschulen festgestellt, dass eine hohe Korrelation zwischen der tatsächlichen Beherrschung eíner Programmiersprache und dem Verständnis von Informatikkonzepten besteht. Wer meint, dass praktische Programmierung „nur“ ein Anhang zur Vorlesung sei, „hat schon verloren“. Die Beschäftigung mit der Informatik bedarf stets der Untermauerung durch konkrete Programmierung, d. h., die Prinzipien und das konkrete Handwerk gehören zusammen.

0.1.3 Umfang und Anforderungen (Anwesenheit reicht nicht!)

Die Veranstaltung umfasst 11 Wochen im Wintersemester 2010 / 2011. Sie besteht aus (eine „Vorlesungs- oder Laborstunde“ entspricht 45 Zeit-Minuten):

- Vorlesung: 3 Vorlesungsstunden pro Woche
- Labor: 4 Laborstunden pro Woche

Wir erwarten von Ihnen regelmäßige Mitarbeit, insbesondere das Einüben und Festigen des Stoffs der Vorlesung durch Nacharbeit und Übungen. Der studentische Aufwand beträgt für den Vorlesungsanteil (und die in diesem Rahmen gestellten Übungsaufgaben) wöchentlich etwa 4 Zeitstunden, d. h. 2.25 Zeitstunden für den Besuch der Vorlesung und 1.75 Zeitstunden zur Nachbereitung des Vorlesungsstoffs und Bearbeitung der Übungsaufgaben (empfohlen werden Lerngruppen mit 3 oder 4 Mitgliedern).

Sehr wichtiger Hinweis: Machen Sie sich einen Plan, wie Sie Ihre Arbeitszeit über die Woche auf Ihre Studienfächer verteilen und halten Sie diesen auch ein!

Die Teilnehmer(innen) sind aufgefordert, den Kontakt zum Dozenten zu suchen. Hierfür können z. B. E-Mails oder die direkte Ansprache in und nach den Lehrveranstaltungen genutzt werden.

Wir bemühen uns darum, alle Hörer(innen) auf eine erfolgreiche Prüfung vorzubereiten. Defizite zeigen sich aber meist schon frühzeitig, vor allem beim Abschneiden in den Übungen / Labor. Dies müssen Sie selbst erkennen! Lassen Sie Ihr Studium nicht schleifen, sondern überwachen Sie sich und suchen Sie den Kontakt zum Dozenten, sobald Ihre Leistungen nicht mehr über den Minimalanforderungen liegen. Die größte Gefahr, im Studium zu scheitern, liegt im schleichenden Ausklinken aus den Veranstaltungen! Denn der Stoff unserer Veranstaltung baut stets auf dem bereits Gelernten auf und daher steigern sich einzelne Versäumnisse schnell zu großen Wissenslücken.

Durch die empfohlene Zusammenarbeit in Vierer-Lerngruppen soll unter anderem auch diesem schleichenden Ausklinken entgegen gewirkt werden.

0.1.4 Voraussetzungen (Wissen / Können aus der Zeit vor dem Studium)

Vorausgesetzt wird mathematisches Abiturwissen. Programmierkenntnisse werden nicht vorausgesetzt. Solche Kenntnisse mögen hilfreich sein, sofern auf bereits Bekanntes zurückgegriffen wird, sie können aber auch gefährlich sein, wenn die Veranstaltung sich in neue Gebiete begibt und die Betreffenden diesen Wechsel nicht nachvollziehen.

Wer keine Programmiervorkenntnisse besitzt, ist daher keineswegs benachteiligt; er / sie muss sich jedoch von Anfang an möglichst gründlich in die Kontroll- und Datenstrukturen der Programmiersprache einarbeiten und Programme alleine zum Laufen bringen. (Lassen Sie sich hier auf keinen Fall von sog. „erfahrenen“ Mitstudierenden helfen, sondern schreiben Sie Ihre Programme selbst!) Weiterhin setzen wir voraus, dass Sie in der Vorlesungszeit mindestens 42, möglichst 45 Stunden pro Woche für Ihr Studium arbeiten.

Nochmals die dringende Empfehlung: Teilen Sie Ihre Zeit gut ein und überwachen Sie dies!

0.1.5 Lernziele

- Grundkonzepte der Informatik:
  - Modellbildung, Abstraktion, Strukturierung
  - Herangehensweisen zur Problemlösung (schrittweise Verfeinerung, Top-Down-Entwurf)
  - Algorithmen: Entwicklung und Bewertung
  - Standardalgorithmen und -datenstrukturen
  - Modellierungskonzept: Endliche Automaten
- Kenntnis einer höheren Programmiersprache (aktiv, nicht nur passiv!), hier : C/C++
- Am Rande gestreift:
  - ungefähre Vorstellung der Computerhardware
  - Aspekte des Software-Engineerings
- auch(!): soziale Kompetenzen, Softskills:
  - Teamarbeit
  - realistische Einschätzung der eigenen Leistung
  - ...

0.1.6 Materialien und Literatur

Selbstständiges Arbeiten ist notwendig! Zusätzlich zu der Anwesenheit in der Vorlesung und Labor müssen Sie zum Nacharbeiten des Vorlesungsstoffes und Bearbeiten von Übungsaufgaben einen Zeitaufwand von etwa 4 Zeitstunden einplanen.

- Erlernen von C++, Programmierpraxis gewinnen
- Nachlesen der in der Vorlesung angesprochenen theoretischen Konzepte
- Literatur: Für diverse Definitionen und Erläuterungen: „Duden Informatik“, vierte Auflage, Bibliografisches Institut, Mannheim, 2007
- Skript zur Vorlesung, aktuelle Hinweise, Ergänzungen $\rightsquigarrow$ `http://www.inf4dhbw.de.vu`

Literaturhinweise zu C/C++ erhalten Sie im Labor, ebenso Hinweise zum Zugang zu den Rechnerräumen und Software.

0.1.7 Allgemeine Hinweise zum Programmieren

Ein Tipp (speziell für Studierende ohne Vorkenntnisse): Schauen Sie sich (leichte) Beispielprogramme in Büchern an und vollziehen Sie nach, was im Programm durch welche Befehle bewirkt wird. Sie können vielleicht Teile daraus wiederverwenden und zur Lösung der hier gestellten Aufgaben benutzen.

Programmieren ist ein „Handwerk“ – man lernt es nur, indem man es selbst durchführt (ein Schreinerlehrling wird das Hobeln einer Tischplatte nie erlernen, wenn er seinem Meister beim Hobeln immer nur zuschaut, er muss es selbst tun, um ein Gefühl dafür zu bekommen). Die Anfangshürde ist dabei immer die schwerste. Jeder muss diese aber selbst überwinden. Wenn Sie nicht weiter kommen, diskutieren Sie mit Ihren Kommillitonen über Ideen zur Lösung. Programmieren Sie diese aber unbedingt selbst aus.

Kommentieren Sie Ihr Programm! Dies heißt insbesondere, dass Sie neben Autor und Datum in den Kommentarzeilen die Idee für Ihr Vorgehen beschreiben sollten und die wesentlichen Bestandteile des Programms beschreiben. Dies ist in der Regel mehr als der Text „Ein schönes C++-Programm“.

Einrückungen machen Programme deutlich lesbarer! Nutzen Sie diese Möglichkeit.

0.1.8 Der Softwareentwicklungsprozess – ein Beispielproblem

Als Beispiel für die Aufgabenfelder der Informatik (auch wenn es für uns in dieser Veranstaltung weit außerhalb unserer Reichweite ist) betrachten wir ein System zur Wettervorhersage. Eine Herausforderung für die „systematische und automatisierte Informationsverarbeitung“: wir haben

> Eingabedaten: eine Flut von Messwerten, die uns die aktuelle Wettersituation und deren bisherige Entwicklung liefern

und sollen daraus

> Ausgabedaten: die Wettervorhersage für morgen

gewinnen. Das Beispiel ist kompliziert genug, dass hier wohl kaum jemand der Versuchung erliegt, sofort zum Computer zu rennen und ein Programm einzutippen – hier ist hoffentlich klar, dass einiges an Vorarbeit geleistet werden muss, bevor wir den Computer einschalten.

Modellbildung: Um unsere Aufgabe mit dem Rechner angehen zu können, müssen wir ein Modell des relevanten Teils der Realität entwickeln – Stichworte sind hier Abstraktion und Formalisierung.

- Für viele Problemstellungen liefert die Mathematik einen mächtigen Satz Werkzeuge zur präzisen Beschreibung der vorkommenden Größen und deren Beziehungen. Im Fall der Wettervorhersage lassen sich etwa Strömungsvorgänge mittels Differentialgleichungen beschreiben.

- Außer Mathematik brauchen wir bei diesem Schritt in der Regel Hilfe aus der Anwendungsdisziplin, hier aus der Meteorologie, aus der Physik und möglicherweise aus weiteren Disziplinen.
- Am Ende der Modellbildung ist aus dem unpräzisen Szenario („das Wetter“) eine präzise Beschreibung (in Größen wie „Temperaturverteilung im Gebiet XY“) geworden, die als Grundlage für die weiteren Arbeiten fassbar ist.

Meist wird in diesem Prozess erst klar, was der Kunde eigentlich will: vielleicht wollte er gar kein Programm, das die Temperaturverteilung vorherberechnet, sondern nur eines, das Bauernregeln auswertet?

Formalisierung und Abstraktion sind wichtige Säulen des Problemlösungsprozesses (auch wenn der Sinn bei den kleinen Beispielen, wie wir sie in den Übungen behandeln können, vielleicht nicht so offensichtlich ist)!

Computergerechte Repräsentation: Bei der Formalisierung hatten wir zwar schon im Hinterkopf, dass mal ein Programm entstehen soll, aber im Mittelpunkt der Überlegungen stand die Beschreibung des Problems, nicht seine Lösung: eine Differentialgleichung ist kein Programm!

Der nächste Schritt bewegt sich in Richtung Rechner. Das formalisierte Problem wird in eine für den Rechner geeignete Form aufbereitet:

- Was für Daten sollen gespeichert werden $\rightsquigarrow$ Datenstrukturen
- Was soll mit diesen Daten passieren $\rightsquigarrow$ Algorithmen (Berechnungsvorschriften)

Auswahl / Entwicklung von Datenstrukturen: Wie sieht eine geeignete Repräsentation der Problemgrößen auf einem Rechner aus?

- Unser Rechner hat z. B. einen zwar großen, aber endlichen Speicher: schon bei der Darstellung (reeller) Zahlen sind Näherungen notwendig.
- Wesentlich problematischer: Feldgrößen (etwa der Temperaturverteilung über einem Gebiet); prinzipiell unendlich viele Funktionswerte. Hier müssen noch mal Mathematiker und Informatiker (und der Kunde) gemeinsam ran, um eine geeignete diskretisierte (endliche) Näherungsdarstellung der verwendeten Größen zu finden.

So mathematiklastig geht es nicht immer zu, die Auswahl geeigneter Datenstrukturen für die verwendeten Größen ist aber fast immer ein wesentlicher Schritt der Programmentwicklung. Das Rad nicht neu erfinden (zumal ein Vorrat von Rädern zur Verfügung steht, auf die wir selber vermutlich nicht kommen würden): Kenntnis von Standard-Datenstrukturen ist hier für den Erfolg sehr wichtig!

Auswahl / Entwicklung von Algorithmen: Mit diesen Daten soll natürlich auch etwas passieren: aus der Aufgabenstellung ist ein Berechnungsverfahren zu gewinnen, ein Algorithmus.

Hierbei sollte man zwar schon an die folgende Implementierungsarbeit denken, aber sich noch nicht mit zu vielen Details (etwa der verwendeten Programmiersprache) befassen.

Implementierung: Ist man sich über Algorithmen und Datenstrukturen im Klaren, muss das ganze in ein Programm(-system) umgesetzt werden, das unser Rechner verarbeiten kann.

Wichtige Frage dabei: was für Werkzeuge gibt es bereits, die uns die Arbeit erleichtern?

- Unumgänglich für praktisch jede Programmentwicklung: Übersetzer einer Hochsprache.

  Dies ermöglicht uns, ein Programm hinzuschreiben, ohne uns um die hässlichen Details der Hardware zu kümmern.

  Die Programmiersprache, die wir in dieser Vorlesung benutzen werden, heißt C/C++.

- Gibt es fertige Programmmodule, die uns unterstützen, z. B.: sollen wir zum Abspeichern unserer Daten ein Datenbanksystem kaufen oder programmieren wir alles selber? Auch hier gilt mal wieder: nicht das Rad neu erfinden! Im Netz sind umfangreiche Programmsammlungen verfügbar, zum Teil (nicht immer!) in sehr guter Qualität.

- Im Beispiel Wettervorhersage sollten wir uns die vorhandenen Programmsysteme zum Lösen von Differentialgleichungen ansehen, auf jeden Fall die vorhandenen Numerikbibliotheken sichten; ziemlich sicher werden wir ein Programm zur Visualisierung dazukaufen.

- Bei einem komplexen Vorhaben wird auch Software nützlich sein, die die Programmentwicklung unterstützt (für unsere Übungsaufgaben wird das keine Rolle spielen): Dokumentation der Programmentwicklung, Unterstützung von Teamarbeit etc. sind Aufgaben, die man nicht unterschätzen sollte.

Was sonst noch zum Entwicklungsprozess dazugehört:

- Analyse des Produktes: Arbeitet es korrekt? Arbeitet es effizient? Kritisches Hinterfragen während aller Entwicklungsschritte!

  Die Formalisierung ermöglicht es, präzise Aussagen dazu zu machen. (Ob die Formalisierung selber korrekt war, ist ein anderes Problem)

- Darstellung und Interpretation der Ergebnisse

- Wartung und Weiterentwicklung

- ...

Der Umfang solch eines Projektes übersteigt das, was wir hier in der Einführungsvorlesung machen können, bei Weitem. Wir werden nichtsdestotrotz diese Konzepte des „Programmierens im Großen“ ($\rightsquigarrow$ Teamarbeit, Organisation, Schnittstellen, ...) in unserem Rahmen („Programmieren im Kleinen“ $\rightsquigarrow$ das, was ein Einzelner in relativ kurzer Zeit alleine umsetzen kann) versuchen anzuwenden.

Diese strukturierte Herangehensweise an Softwareprojekte ist unter dem Namen Wasserfall-Modell bekannt. Es setzt sich aus folgenden Phasen zusammen.

1. Analyse (Zusammentragen der Wünsche des Kunden und Analyse des Ist-Zustands)
2. Spezifikation (Darstellung des Soll-Zustands, Verpflichtung gegenüber dem Kunden)

3. Entwurf (Planung der Datenstrukturen, Module der Software)
4. Implementierung (Umsetzung des Entwurfs in eine Programmiersprache)
5. Test
6. Auslieferung und Wartung

Der Schwerpunkt dieser Vorlesung liegt auf dem Entwurf und der Implementierung.

1 Programmierung – eine Einführung in den strukturierten Entwurf und dessen Implementierung in C/C++

Vieles von dem, was wir tun, läuft unbewusst ab. Ein Großteil dessen läuft in Wiederholungen und Fallunterscheidungen ab. Fahren wir z. B. Auto, so wiederholen wir ständig für jede von außen eintreffende Information einen Prozess, der abhängig von dieser Information entweder eine Handlung einleitet (in den nächsthöheren Gang schalten oder bremsen oder blinken oder ...) oder auch die Information für unwichtig erachtet und einfach nichts tut. Wir werden im ersten Semester unter anderem lernen müssen, zu abstrahieren und uns zu überlegen, wie die zu modellierenden Prozesse in Wiederholungen und Fallunterscheidungen ablaufen. Dieses Strukturieren ist ein Prozess, den man – wie das Programmieren selbst auch – am besten durch Üben erlernt. In diesem Sinne ist diese Vorlesung nur ein Leitfaden zu den Konzepten. Das Üben müssen Sie selbst übernehmen.

Programmieren heißt **Modellieren** (abstrahieren und formalisieren), **Strukturen erkennen** (im Ablauf und den zu verabeitenden Daten der modellierten Aufgabe) und diese in eine **Programmiersprache** umsetzen.

1.1 Pseudo-Code und Nassi-Shneiderman-Diagramme

An dem folgenden Beispiel wollen wir zwei weit verbreitete Darstellungsformen einführen: Pseudo-Code und Nassi-Shneiderman-Diagramme (auch bekannt als Struktogramme).

1.1.1 Das Spiel Mensch-Ärgere-Dich-Nicht

Dieses Spiel wird uns im Laufe des Semesters immer wieder begleiten. Wir rekapitulieren zunächst einige Regeln ... (zitiert nach `http://de.wikipedia.org/wiki/Mensch_%C3%A4rgere_dich_nicht`)

- Wer eine Sechs würfelt, muss eine eigene Spielfigur aus der Startposition heraus auf sein Startfeld des Spielfeldes stellen (auch wenn er mit einer anderen Figur einen ihm nützlicheren Zug machen könnte). Danach darf er erneut würfeln und mit der Figur entsprechend viele Felder vorrücken.
- Das Startfeld muss so bald wie möglich wieder freigemacht werden.
- Hat er aber keine Figur mehr in der Startposition, so steht es ihm frei, die erwürfelten sechs Felder mit einer Figur seiner Wahl vorzurücken. Auch dann darf er erneut würfeln und einen weiteren Zug machen.

- Kommt beim Umlauf eine Spielfigur auf ein Feld, das bereits von einer gegnerischen Spielfigur besetzt ist, gilt die gegnerische Figur als geschlagen und muss zurück auf ihre Startposition. Eigene Figuren können nicht geschlagen werden - steht eine eigene Figur auf dem Zielfeld, ist der Zug unausführbar.
- Hat ein Spieler mehrere Spielfiguren im Umlauf, kann er entscheiden mit welcher er ziehen möchte.
- Ein Würfelwurf darf allerdings nicht aufgeteilt werden.
- Hat ein Spieler überhaupt keine Figur auf dem Spielfeld (was bei Spielbeginn natürlich alle Spieler betrifft), so hat er in jeder Runde drei Versuche, die nötige Sechs zu würfeln, um eine Figur ins Spiel zu bringen.
- Optionale Regeln: Wer mit einer gewürfelten Zahl eine gegnerische Figur schlagen kann, muss dies tun (Schlagzwang). Übersieht er es und zieht eine andere eigene Figur, so dürfen die Gegner eine Figur des Spielers „pusten“, d. h. zurück in die Startposition stellen.
- Im Zielbereich darf man eigene Figuren nicht überspringen.

Bei der Entwicklung von Programmen wollen wir jedoch nicht Regeln aufstellen – Algorithmen sind Verabeitungsvorschriften, d. h., sie beschreiben präzise und eindeutig schrittweise den Ablauf. Beginnt man, den Ablauf dieses Spiels zu beschreiben, so könnte dies z. B. wie folgt aussehen:

```
Auswürfeln, wer beginnt
solange noch keiner gewonnen hat (d.h., alle Figuren in seinem Zielbereich hat)
  bestimme den nächsten Spieler, der am Zug ist
  falls der Spieler noch keine Figur auf dem Spielfeld hat:
    wiederhole maximal 3 Mal bis eine 6 gewürfelt wurde
      falls 6 gewürfelt
        setze Figur auf die Startposition
        falls dort eine gegnerische Figur steht
          schlage diese
        würfele nochmal
        falls 6 gewürfelt
          setze die Figur vom Startfeld 6 weiter
          würfele nochmal
          falls 6 gewürfelt
            setze (zweite) Figur auf die Startposition
            würfele nochmal
            falls 6 gewürfelt
              ...
```

Man sieht schnell, dass man so mit den fortgesetzen Fallunterscheidungen kein Ende findet (spätestens gegen Ende des Spiels kann es passieren, dass der Spieler prinzipiell unendlich oft hintereinander eine 6 würfeln kann, die letzte Spielfigur aber nur noch ein Feld vor dem letzten freien Platz im Zielbereich steht).

Hier stehen wir nun vor der Aufgabe, die Strukturen im Ablauf zu erkennen. Versuchen wir zunächst die „Grobstruktur“ zu beschreiben und das Problem in handhabbarere Teilprobleme

zu zerlegen. Diese könnten hier z. B. das „Auswürfeln, wer beginnt“, „Spielzug, wenn 6 gewürfelt wurde“ und „Spielzug, wenn keine 6 gewürfelt wurde“ sein. Der Gesamtzug eines Spielers lässt sich dann z. B. wie folgt beschreiben

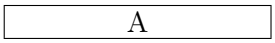
```
falls keine (eigene) Spielfigur auf dem Feld ist
  wiederhole
    würfeln
  bis 3 mal gewürfelt wurde oder eine 6 gewürfelt wurde
  falls 6 gewürfelt wurde
    wiederhole
      "Spielzug, wenn 6 gewürfelt wurde"
    bis keine 6 mehr gewürfelt wird
    "Spielzug, wenn keine 6 gewürfelt wurde"
sonst
  solange 6 gewürfelt wird
    "Spielzug, wenn 6 gewürfelt wurde"
  "Spielzug, wenn keine 6 gewürfelt wurde"
```

Die in `"..."` gesetzten Anweisungen sind jedoch für sich noch nicht selbsterklärend und müssen näher spezifiziert werden (selbst durchführen).

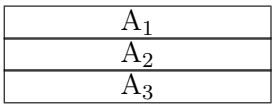
Bereits 1966 haben Böhm und Jacopini gezeigt, dass jeder Algorithmus ausschließlich mit Sequenz, Fallunterscheidung und Wiederholung dargestellt werden kann. 1968 führte E. W. Dijkstra dann den Begriff der Strukturierten Programmierung ein, Kern ist dabei, dass Programme, die sich auf obige Strukturen beschränken (insbesondere keine beliebigen Sprünge im Programm zulassen) leichter verständlich und nachvollziehbar sind.

Dies führte 1972/73 zur Entwicklung der Darstellung von Algorithmen als Nassi-Shneiderman-Diagramme (Struktogramme):

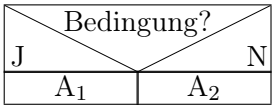
- jede (elementare) Anweisung als Rechteck



- Sequenzen von Anweisungen

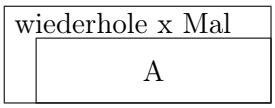


- Fallunterscheidungen

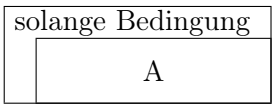


- Bei Schleifen unterscheidet man Zählschleifen (bei diesen ist die Anzahl der Wiederholungen vorab bekannt) und bedingte Schleifen (Wiederholung solange eine Bedingung gilt (kopfgesteuerte Schleife) oder bis eine Bedingung gilt (fußgesteuerte Schleife))

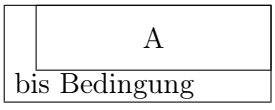
  Zählschleifen



  Kopfgesteuerte Schleifen



  Fußgesteuerte Schleifen



Übung: Überführen Sie den obigen Pseudo-Code in ein äquivalentes Struktogramm

Während Struktogramme nach DIN 66261 genormt sind und so in der Literatur weitgehend einheitlich dargestellt werden, gibt es keine festen Regeln für Pseudo-Code. Um die Übersichtlichkeit zu wahren, sollten Sie auch hierbei die Strukturen durch Einrückungen klarstellen. Zum Teil wäre es ohne Einrückung auch nicht eindeutig (im obigen Beispiel eines Zuges könnte sich das `sonst` auch auf das zweite `falls` beziehen – überlegen Sie sich selbst die Konsequenzen)

1.1.2 Übungsbeispiel: Winterreifenwechsel

Entwerfen Sie einen Algorithmus (als Struktogramm), der den Wechsel von Sommer- auf Winterreifen beschreibt. Machen Sie sich dabei insbesondere die Problematik klar, was als elementare Anweisung angesehen werden kann, die ohne weitere Erläuterung für jeden verständlich ist: ist „Schraube lösen und herausdrehen“ bereits eine elementare Anweisung oder muss diese noch näher erklärt werden („Drehkreuz auf Schraubenkopf setzen und Drehkreuz solange nach links drehen bis Schraube draußen ist“)?

Wenn wir später Algorithmen entwerfen und in C/C++-Programme umsetzen, müssen wir uns bewusst sein, welche Operationen die Hochsprache C/C++ bereits zur Verfügung stellt und welche wir aus einfachereren Operationen selbst zusammensetzen müssen.

1.2 Definition Algorithmus und Programm

Die Begriffe Algorithmus und Programm wollen wir nun genauer fassen.

1.2.1 Ein Algorithmus ist ...

- eine Verarbeitungsvorschrift, die schrittweise abgearbeitet wird. Dabei werden in jedem Schritt
  - nur endlich viele Operationen ausgeführt,
  - die jede nur auf endlich viele Daten zugreift und
  - jede nur endlich viele Daten erzeugt.

  Die Verarbeitungsvorschrift selbst (also der Text, der die Vorschrift beschreibt) ist endlich lang.
- präzise formuliert
- von jedem ohne weitere Erläuterungen durchführbar
  - je nach Kontext kann zur Durchführung noch weiteres Wissen notwendig sein – dieses ist dann aber eindeutig und nicht spezifisch für den Algorithmus, z. B. „bilde zu $f(x)$ die erste Ableitung $f'(x)$“
- enthält der Algorithmus umgangssprachliche Elemente, so müssen diese eindeutig interpretierbar sein
- es gibt beschränkt viele Basisoperationen, die jeweils in konstanter Zeit ausführbar sind.

Oft wünscht man noch, dass die Vorschrift deterministisch sein soll, d. h., in jedem Zustand ist eindeutig bekannt, welches der nächste auszuführende Schritt ist.

Kochrezepte haben i. A. nicht die Eigenschaft eines Algorithmus, da „eine Prise Salz“ oder „ein wenig Öl“, ... nicht wirklich eindeutig sind.

Es ist nicht entscheidend, ob ein Algorithmus irgendwann anhält. Zur Ausgabe aller natürlichen Zahlen können wir folgenden Algorithmus angeben:

1. merke dir die Zahl 0
2. gebe die gemerkte Zahl aus
3. erhöhe die gemerkte Zahl um 1 und merke dir nun diese (und nur diese) Zahl
4. gehe zu Schritt 2

Dieser hat eine endliche (sogar sehr kurze) Darstellung. Gibt man hingegen alle Zahlen explizit aus wie hier

1. gebe die Zahl 0 aus
2. gebe die Zahl 1 aus
3. gebe die Zahl 2 aus
4. gebe die Zahl 3 aus

⋮ ⋮

– so ist dies kein Algorithmus, da die Darstellung nicht endlich ist. In der Praxis wären solche Berechnungsvorschriften mangels Möglichkeit zur Speicherung auf externen Datenträgern sowieso ungeeignet.

In der Praxis ist man in der Regel an Algorithmen interessiert, die für jede Eingabe nach endlich vielen Schritten die Berechnung abgeschlossen haben (Ausnahmen sind z. B. Betriebssysteme oder Steuerungssysteme).

Umgangssprachlich können wir Algorithmus mit dem Begriff „eindeutiges Kochrezept“ übersetzen.

Im Allgemeinen gibt es für ein gegebenes Problem mehr als einen Algorithmus (sogar unendlich viele).

1.2.2 Was unterscheidet ein Programm von einem Algorithmus?

- eindeutiger Formalismus einer Programmiersprache mit vorher festgelegtem Befehlssatz mit vorher festgelegter Struktur (die Syntax der Programmiersprache)
- Bezug auf bestimmte Darstellung der verwendeten Daten
- Schnittstellen zu anderen Programmen (z. B. Betriebssystem) und Hardware
- Ausführbarkeit auf einem Computer

Mit der Umsetzung eines Algorithmus in ein auf einem Computer ausführbaren Programm kommen in der Regel noch die beiden folgenden Eigenschaften bzw. Einschränkungen dazu

- Näherungen (z. B. bei reellen Zahlen)
- endlicher Speicher

Ein Algorithmus kann in verschiedene Programme umgesetzt werden (verschiedene Programmiersprachen, verschiedene Betriebssysteme, verschiedene Computerhardware). Ein Algorithmus ist somit die abstrakte Formulierung aller Programme, die ihn beschreiben.

1.2.3 Übungsbeispiel: Ist das ein Algorithmus?

Begib dich zum Südwest-Eingang des Hauses in der Jägerstraße 58. Warte bis Mitternacht. Peile den äußersten Deichselstern des Sternbilds des großen Wagen an und gehe dann genau 107 Schritte in diese Richtung. Grabe hier und du stößt in 2 Ellen Tiefe auf den Schatz.

Ist die Vorschrift eindeutig? Welche Informationen fehlen?

1.3 Die Rollenspiel-Metapher

1.3.1 Programmierer, Ausführer und Benutzer

Vieles in der Informatik wird formal und mathematisch beschrieben. Unter anderem zur Motivation, warum überhaupt eine formale Beschreibung sinnvoll ist, betrachten wir, welche Rollenverteilung es beim Programmieren im Allgemeinen gibt.

- Programmierer: schreibt Programme (in einer Hochsprache); er alleine ist dafür verantwortlich, was das von ihm geschriebene Programm leistet (Semantik (= Bedeutung) des Programms) – hier: Sie!
- Ausführer: – hier: Übersetzer/Compiler von C++ (+ Windows / Linux)
  - prüft die Programme auf syntaktische Korrektheit (d. h., er prüft, ob das Programm ein für den Rechner formuliertes „detailliertes, eindeutiges Kochrezept“ ist, ob der formale Aufbau den Regeln der Hochsprache genügt)
  - führt das Programm (falls es syntaktisch korrekt ist) mechanisch Schritt für Schritt nach dem „Kochrezept“ aus; der Ausführer hat keine Vorstellung davon, was der Programmierer sich gedacht haben könnte – er tut genau das und nur das, was im Programm steht
- Benutzer: – hier: Sie!
  - lässt den Ausführer Programme ausführen
  - ist dabei für Eingaben und insbesondere Interpretation(!) der Ausgaben zuständig

Notwendige Kenntnisse für den Programmierer

- Syntax und Semantik der Programmiersprache

- Problemstellung, welche Eingaben sollen zu welchen Ausgaben verarbeitet werden
- ggf. Kreativität und Genialität (oder Expertenwissen von außen), wenn effiziente Lösungen oder ähnliches gefragt sind

Notwendige Kenntnisse für den Ausführer

- Zur Überprüfung, ob es sich um ein Programm handelt: Syntax der Programmiersprache
- Voraussetzung: Eindeutige Beschreibungen $\rightsquigarrow$ Formale Darstellung (speziell, wenn das Ausführen mechanisch durchgeführt wird) $\rightsquigarrow$ Erweiterte Backus-Naur-Form (EBNF), Syntaxdiagramme (siehe nächster Abschnitt)
- Zur Ausführung des Programms wird interessanterweise die Semantik der Programmiersprache nicht benötigt. Der Übersetzungsprozess von der Hochsprache in Maschinencode kann automatisch erfolgen, ohne dass der Compiler wissen muss, was die einzelnen Bausteine bedeuten. Er übersetzt lediglich nach bestimmten Regeln Ihr Programm in ein Muster aus 0en und 1en. Die Semantik des Maschinencodes ist in der CPU fest verdrahtet und kann nicht verändert werden – daher muss diese auch nicht formal beschrieben werden.

  Führt man das Programm von Hand aus, muss die Semantik dem Ausführenden natürlich bekannt sein. Genauso muss derjenige, der den Compiler programmiert hat, die Semantik gekannt haben, um die für die Umsetzung von der Hochsprache in Maschinencode notwendigen Regeln aufstellen zu können.

Notwendige Kenntnisse für den Benutzer

- Keine, wenn das Programm entsprechend geschrieben ist

Das Programm aus Sicht des Ausführers

- Eingabe für den Ausführer: hier: eine Textdatei mit dem Programm in C/C++
- 1. Aufgabe: Analyse des Programms, Zerlegung des Textes in Lexikalische Einheiten (Bausteine des Programms wie z. B. Schlüsselwörter, Zahlen, Operatoren, Kommentare, Trennzeichen)
- 2. Aufgabe: Überprüfung auf formal korrekten Aufbau
- ggf. Rückmeldung an den Programmierer $\rightsquigarrow$ dieser muss sich über die lexikalischen Einheiten bewusst sein, um die Fehlermeldungen des Ausführers verstehen zu können
- 3. Aufgabe: Ausführen des Programms

Formale Syntax ist auch für den Programmierer sinnvoll, da diese den strukturellen Aufbau der Sprache verdeutlicht.

1.3.2 Compiler und Interpreter

Das Ausführen eines Programms durch einen *Compiler* und das Betriebssystem hat typischerweise die Eigenschaft, dass der Programmtext in der Hochsprache zunächst komplett betrachtet, dann in Maschinencode umgesetzt und schließlich vom Betriebssystem durch die CPU ausgeführt wird. Beim eigentlichen Ausführen hat der Compiler keinen Einfluss mehr. Dies unterscheidet sich grundlegend von dem, wie wir als Mensch unseren Programmcode von Hand ausführen würden. Hier wird der Code Zeile für Zeile betrachtet und jede direkt ausgeführt (ohne davor den Rest bereits gelesen zu haben). Dabei können wir den Kontrollfluss jederzeit unterbrechen, um uns z. B. über die Belegung von Variablen (Wertebehältern) klar zu werden, und danach (der Semantik der Programmiersprache folgend) im Programm fortzufahren. Syntaxfehler werden so erst an der entsprechenden Stelle beim Ausführen erkannt (der Compiler hätte aufgrund dieser den Programmtext erst garnicht in Maschinencode übersetzt). Diese Möglichkeiten wie beim Ausführen von Hand bietet ebenso das Ausführen eines Programms durch einen *Interpreter*. In der Regel ist die Ausführungsgeschwindigkeit von Programmen mithilfe von Compilern deutlich schneller als mit Interpretern. Für die Sprache C/C++ gibt es fast ausschließlich Compiler – bei anderen Programmiersprachen wie z. B. Prolog sind Interpreter die Norm.

1.4 Sprachen zur Beschreibung der Syntax von Sprachen

Der wesentliche Aufbau von Programmen in C/C++ (und der meisten anderen Hochsprachen) lässt sich relativ leicht beschreiben. Wir stellen hier zwei Möglichkeiten vor

- (E)BNF - (Erweiterte) Backus-Naur-Form
  - wird in den meisten Lehrbüchern verwendet
- Syntaxdiagramme
  - sind etwas anschaulicher, aber in der Praxis auch etwas umständlicher zu handhaben

Hier sollen zunächst nur die wenigen Bausteine der EBNF und der Syntaxdiagramme vorgestellt werden, so dass wir damit die Syntax von C/C++ formal beschreiben können und Sie die Darstellung in Lehrbüchern verstehen.

Welche weitergehenden Eigenschaften die EBNF und Syntaxdiagramme haben, was genau damit beschrieben werden kann und wo die Grenzen der Möglichkeiten liegen, werden wir hier nicht betrachten. Es reicht zu wissen, dass alles, was mit EBNF dargestellt werden kann, auch mit Syntaxdiagrammen möglich ist und umgekehrt.

So, wie wir in der Hochsprache Folgen von Anweisungen, Wiederholungen und Fallunterscheidungen programmieren, sind auch bei der formalen Beschreibung der Syntax diese Grundkonstrukte Sequenz, Wiederholung und Fallunterscheidung nötig.

Wir führen hier nun an dem Beispiel einer Mehrfach-Fallunterscheidung mit `if-elseif-else` vor, wie sich dessen Syntax mittels Syntaxdiagramm und EBNF darstellen lässt.

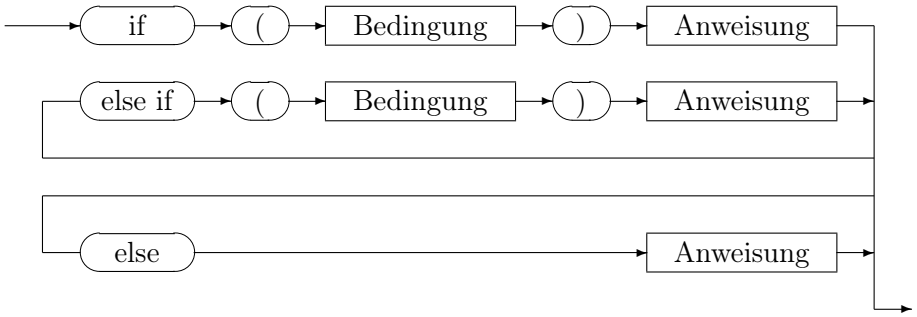
Darzustellen ist folgender Aufbau: Eine `if-elseif-else`-Anweisung beginnt stets mit `if`, gefolgt von einer in runden Klammern gesetzten Bedingung (Boole'scher Ausdruck = Ausdruck, der sich zu wahr oder falsch auswertet). Dann kommt eine Anweisung. Optional kommen dann

beliebig viele (oder auch garkeine) `else-if`-Zweige (jeder Zweig beginnt mit dem Schlüsselwörtern `else if`, gefolgt von einer in runden Klammern gesetzten Bedingung und einer Anweisung. Optional kann dann noch ein `else`-Zweig folgen (Schlüsselwort `else` plus eine Anweisung).

In EBNF formulieren wir das wie folgt – wir geben die EBNF-Regeln gleich so an, dass auch sinnvolle Einrückungen mit dargestellt werden:

```
<if-elseif-else>  ::=    "if" "(" <Boolescher-Ausdruck> ")"
                              <Anweisung>
                         { "else if" "(" <Boolescher-Ausdruck> ")"
                              <Anweisung> }
                         [ "else"
                              <Anweisung> ]
```

Hier werden also Iterationen und optionale Elemente mit geschweiften bzw. eckigen Klammern dargestellt. In Syntaxdiagrammen werden diese durch Verbindungen mit Pfeilen modelliert.



Es sei hier nochmals betont, dass es dabei um die Syntax (also den formalen Aufbau) des `if-elseif-else`-Konstrukts geht. Die Semantik, also die Bedeutung für den Ablauf in einem Programm, z. B., dass die Anweisungen im „`wenn-wahr`“-Zweig nicht ausgeführt werden, wenn die Bedingung sich zu `false` auswertet, ist etwas anderes. Die Syntax besagt nur, dass nach der Bedingung stets eine Anweisung stehen muss, nicht dass diese auch in jedem Fall ausgeführt wird.

Genau genommen gibt es in C/C++ nur eine `if-else`-Anweisung mit EBNF-Regeln

```
<if-else>  ::=    "if" "(" <Boolescher-Ausdruck> ")"
                       <Anweisung>
                  [ "else"
                       <Anweisung> ]
```

Die obige fiktive `if-elseif-else`-Anweisung entspricht der `if-else`-Anweisung, bei der im „`else`“-Zweig jeweils wieder eine `if-else`-Anweisung steht. Wir haben oben gesagt, dass der „`else`“-Zweig optional ist, also auch weggelassen werden kann. Dies führt zu Mehrdeutigkeiten, wenn `if`-Anweisungen geschachtelt werden, die zum Teil keinen „`else`“-Zweig haben.

Untersuchen Sie folgendes Programmfragment. Wo ist die Mehrdeutigkeit (finden Sie Anweisungen und Bedingungen, die zu verschiedenen Ergebnissen führen)? Wie löst C/C++ diese Mehrdeutigkeit auf?

```
if (Bedingung1)
if (Bedingung2)
   Anweisung1;
else
   Anweisung2;
```

Diese Art von Mehrdeutigkeit kann vermieden werden, wenn die Anweisungen in beiden Zweigen jeweils in geschweifte Klammern – also als Anweisungsblock – gesetzt werden (selbst überlegen). Sie sollten daher hiervon bei `if`-Anweisungen immer Gebrauch machen; für eine einheitliche Darstellung auch dann, wenn nur eine einzelne Anweisung ausgeführt werden soll, bei der die geschweiften Klammern nicht unbedingt nötig wären.

1.4.1 EBNF

Hier noch einmal das Wesentliche über die EBNF kurz zusammengefasst:

- Formale Beschreibungssprache für Zeichenketten
- Terminale (direkt aufgeschriebene Zeichenketten): z. B. `"Text"`, `"42"`, ... – in Anführungszeichen (häufig findet man auch Varianten ohne Anführungsstriche)
- Nichtterminale (Variablen): z. B. `<if-elseif-else>`, `<Ausdruck>`, `<Ziffer>`, ... – in spitzen Klammern
- „wird definiert durch“: „`::=`“
- Alternativen: z. B. `<Ziffer> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"` – Trennung durch senkrechten Strich
- Optionale Elemente: z. B. `[ "else" <Anweisung> ]`, ... – in eckigen Klammern, 0 oder 1 Mal
- Iterationen: z. B. `{ "else if" "(" <Boolescher-Ausdruck> ")" <Anweisung> }` – in geschweiften Klammern, 0, 1, 2, ... Mal
- Zur Gruppierung von Elementen können runde Klammern verwendet werden: z. B. `<Zahl> ::= ("+"|"-") {<Ziffer>}` (nach diesem Beispiel wäre eine Zahl nur Zahl, wenn sie stets ein Vorzeichen hat – außerdem dürfte die Zahl führende 0en haben)

Es gibt in der Literatur diverse Variationen davon (z. B. Markierung des Endes einer Regel durch einen Punkt, `(..)`* statt `{..}`, ...).

1.4.2 Syntaxdiagramme

Hier noch einmal das Wesentliche über die Syntaxdiagramme kurz zusammengefasst:

- Formale Beschreibungssprache für Zeichenketten
- Terminale (direkt aufgeschriebene Zeichenketten): z. B. Text, 42, ... – in Kreisen (oder Ellipsen)

- Nichtterminale (Variablen): z. B. if-Anweisung, Ausdruck, Ziffer, ... – in Rechtecken
- „wird definiert durch“: Name steht über dem Diagramm
- Kreise und Rechtecke werden durch Pfeile verbunden
- Alternativen, optionale Elemente, Iterationen: durch Aufsplittung der Pfeile / Verbindungen – die durch ein Syntaxdiagramm definierten Zeichenfolgen ergeben sich durch alle möglichen Reihenfolgen vom Anfang zum Ende des Syntaxdiagramms zu gelangen.

Da es zu jedem Konstrukt in der EBNF eine Darstellungsform als Syntaxdiagramm gibt, ist klar, dass man mit Syntaxdiagrammen mindestens so viel darstellen kann wie mit der EBNF. Es lässt sich aber auch zeigen, dass man die durch Syntaxdiagramme dargestellten Zeichenketten stets auch mit EBNF-Regeln angeben kann. Dies ist zuweilen etwas knifflig aber stets möglich. Versuchen Sie einen Algorithmus zu beschreiben, der dieses leistet.

1.5 Variablen und Datentypen

Die Platzhalter in Programmen nennt man wie in der Mathematik *Variablen* (Wertebehälter). Anders als in der Mathematik sind Gleichungen jedoch als Zuweisungen zu lesen (der Wert, zu dem sich die rechte Seite auswertet, wird der Variablen auf der linken Seite zugewiesen – insb. muss links stets eine einzelne Variable stehen). Es sind keine Formeln, die ab diesem Zeitpunkt gelten. Nach der Anweisungsfolge `x=3; y=x+5; x=7;` hat die Variable `x` den Wert `7`, die Variable `y` hat jedoch den Wert `y=3+5=8` und wird durch die Zuweisung `x=7` nicht verändert.

Jede Variable belegt im Speicher Platz. Es ist naheliegend, dass Daten von verschiedener Art verschieden viel Speicher benötigen und auch intern im Speicher verschieden dargestellt sind. Es ist somit hilfreich, wenn wir jeder Variablen einen *Datentyp* zuweisen, sodass der Compiler weiß, wieviel Speicher er für die Variable zur Verfügung stellen muss und wie er den Inhalt der Variablen im Speicher darstellen muss. Datentypen dienen zunächst also einmal zur Strukturierung des Speichers. Daten werden im Rechner ausschließlich durch 0en und 1en, Bits genannt, dargestellt; 8 Bits werden zu einem Byte zusammengefasst und auf den heute üblichen Rechnern 32 Bits zu einem Wort.

Standarddatentypen In C/C++ heißt der Standarddatentyp für ganze Zahlen `int` (von engl. integer). Für eine Variable von diesem Typ werden vom Compiler 32 Bits im Speicher reserviert. 32 Bits erlauben die Darstellung von 2^{32} verschiedenen Werten – im Fall des Datentyps `int` sind dies die ganzen Zahlen von -2^{31} bis $2^{31} - 1$. Zu jedem Datentyp gehört ein *Wertebereich*. Neben Variablen haben auch Konstanten sowie die Ergebnisse von Funktionen einen Datentyp.

Bevor eine Variable verwendet werden kann, muss sie deklariert werden (dem Compiler bekannt gemacht werden). Dabei wird in C/C++ auch der Datentyp festgelegt. C/C++ hat somit ein statisches Typsystem, d. h., zur Übersetzungszeit ist für jede Variable deren Typ bekannt.

Funktionen erwarten Parameter von bei der Deklaration festgelegtem Typ und liefern Ergebnisse, deren Typ ebenfalls bei der Deklaration festgelegt werden.

Die erlaubten Operationen sind abhängig vom Datentyp.

Vorteile einer Programmiersprache mit Typsystem sind die Strukturierung des Speichers, die dem Compiler erlaubt effizienteren Code zu generieren. Außerdem werden durch ein Typsystem

manche Tippfehler zu formalen Fehlern, die der Compiler dann bereits bei der Übersetzung erkennen kann.

Für Zahlen wird zwischen den Datentypen für ganze Zahlen (`int`, mit und ohne Vorzeichen (`unsigned int`, `signed int`), mit 16 Bit Genauigkeit (`short int`), mit 32 Bit Genauigkeit (`long int`, ist auf heutigen 32-Bit-Rechnern identisch mit `int`) – C/C++ erlaubt zwar `int` bei den genauer spezifizierten Varianten wegzulassen, dies wird jedoch aus Gründen der Lesbarkeit nicht empfohlen) und Fließkommazahlen (Zahlen mit variablem Vor- und Nachkommaanteil, Datentyp `float` und mit höherer Genauigkeit `double`) unterschieden. Für Wahrheitswerte (z. B. Ergebnisse von Vergleichstest, also die Werte `true` und `false`) gibt es den Datentyp `bool` (nur in C++), für Zeichen den Datentyp `char`. Die Verwendung werden wir in den folgenden Kapiteln an Beispielen sehen. Auf die rechnerinterne Darstellung gehen wir hier nicht genauer ein.

Erlaubte Operationen auf Zahlen sind neben den Grundrechenarten auch die Restbestimmung bei der ganzzahligen Division (Operator `%`), zum Vergleich von Zahlen dienen die Operatoren <, <=, == (Test auf Gleichheit), ! = (Test auf Ungleichheit), >= und >. Mehrere Wahrheitswerte können über logische Operatoren (&& (logisches und, wertet sich zu wahr aus, wenn beide Operanden wahr sind), || (logisches oder, wertet sich zu wahr aus, wenn wenigstens ein Operand wahr ist), ! (Negation, kehrt wahr zu falsch um und umgekehrt)) zu komplizierten logischen Ausdrücken verknüpft werden.

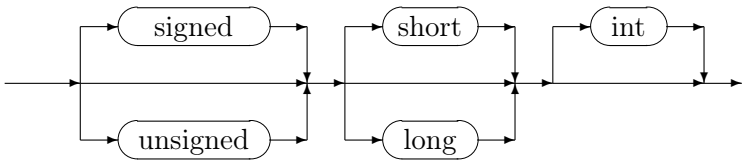
Das Typkonzept von C/C++ ist nicht besonders streng. Dies beschleunigt zwar zum einen den Entwicklungsprozess, da C/C++ viele Typkonvertierungen (sogenannte Casts) zwischen verschiedenen Zahltypen automatisch durchführt. Leider führt dies auch zu schwer zu findenden Fehlern, da C/C++ mit den nicht immer kompatiblen Wertebereichen sehr schlampig umgeht.

Sind z. B. die Datentypen der beiden Operanden einer Addition, Subtraktion, Multiplikation oder Division nicht identisch, so wird vom Compiler eine Typkonvertierung bei einem der Operanden durchgeführt (die oben genannten Operatoren sind eigentlich jeweils nur für zwei Operanden desselben Typs definiert). Dabei wählt der Compiler die ihm am sinnvollsten erscheinende Variante aus. Dies muss nicht zwangsweise mit der Intention des Programmierers übereinstimmen. C/C++ erlaubt explizite Typkonvertierungen. In der klassischen C-Variante wird dies durch Voranstellen des Datentyps erreicht (z. B. `(float)x` zur Umwandlung der Variablen `x` in eine Gleitkommazahl) oder (in C++) in einer funktionalen Notation (z. B. `float(x)`), die auch in anderen Programmiersprachen üblicher ist.

Empfehlung: Überlassen Sie speziell bei Rechnungen die Typkonvertierung nicht dem Compiler. Durch die expliziten Typkonvertierungen vermeiden Sie Mehrdeutigkeiten und machen das Programm dadurch lesbarer und wartbarer.

Schreiben Sie sich ein kleines Testprogramm, das einfache Operationen mit zwei Operanden mit unterschiedlichen Datentypen durchführt (z. B. Multiplikation eines `unsigned int` mit einem `float`), kombinieren Sie dabei auch positive und negative Zahlen. Das Ergebnis hängt unter Umständen auch vom Compiler ab, nach welchen Regeln dieser die impliziten (automatischen) Casts durchführt.

Zur Wiederholung der Syntaxdiagramme noch eine kleine Übungsaufgabe: Bei der Deklaration von Ganzzahltypen kann das `int` entfallen, wenn dies z. B. durch `short` näher definiert wurde. Folgendes Syntaxdiagramm gibt die Möglichkeiten für Ganzzahltypen an.



Dabei ist ein kleiner Fehler unterlaufen: Neben allen erlaubten Kombinationen kann man in obigem Syntaxdiagramm das `int` auch überspringen, wenn weder die Länge der Zahl noch näher definiert wurde, ob die negativen Zahlen ausgeschlossen sind. Finden Sie ein korrektes Syntaxdiagramm.

Randbemerkung: Der neueste Standard der C-Sprache sieht auch einen größeren Ganzzahl-Datentyp `signed/unsigned long long int` vor, der (mindestens) 64 Bit lang ist. In C++ ist dies noch nicht im Sprachstandard verankert, wird also unter Umständen nicht von jedem Compiler unterstützt. Bei manchen Compilern (z. B. von Microsoft) heißt dieser Datentyp in C++ `__int64`, was wiederum nicht von allen anderen Compilern verstanden wird.

1.6 Vom Problem zum Programm

Algorithmen (und deren Umsetzung im Rechner, die Programme) beschreiben in erster Linie, welche Anweisungen in welcher Reihenfolge ausgeführt werden sollen. Beim Entwurf von Programmen müssen wir uns neben dem Programmablauf auch bewusst über die Daten und deren Verarbeitung werden. Ein prinzipielles Vorgehen beim Entwurf von Programmen ist, das Gesamtproblem zunächst in (voneinander unabhängige) Teilprobleme zu zerlegen. Die Teilprobleme sind in der Regel zu komplex, als dass wir diese mit einem oder zwei Befehlen der Hochsprache erledigen könnten, d. h., jedes Teilproblem zerfällt in mehrere Schritte, die von der Komplexität und Abstraktionsstufe einfacher zu handhaben sind. Sind diese Schritte immer noch zu komplex, müssen sie weiter verfeinert werden. Dies führt zu einer Hierarchie der Abstraktionsstufen, über die wir uns beim Programmentwurf bewusst sein sollten.

Ähnliches gilt für die Daten: stellen wir uns vor, wir sollen ein Programm zur Verwaltung der Studierenden eines Jahrgangs verwalten, so bilden auch die zu verwaltenden Daten eine Hierarchie: Der Jahrgang teilt sich in 4 Gruppen auf, in jeder Gruppe sind zwischen 27 und 32 Studierende, für jeden Studierenden sind der Vorname (zusammengesetzt aus bis zu 20 Buchstaben), der Nachname (bis zu 30 Buchstaben), Straße, Hausnummer, Postleitzahl, Ort, Telefonnummer und die Noten zu speichern, letztere einzeln für jede Vorlesung, wobei dabei noch zu unterscheiden ist, ob die jeweilige Prüfung im ersten Versuch oder bei der Wiederholung bestanden wurde.

Zu Beginn jeder Problemstellung wiederum stellt sich die Frage, aus welchen Eingaben welche Ausgaben berechnet werden (und welche Zwischenergebnisse dabei auftreten und wie diese zu der Lösung zusammengefügt werden). Betrachtet man so den Verlauf aus Sicht der Daten, spricht man vom Datenfluss.

Dies sind nun also die 4 Sichtweisen auf Programme und deren Daten, wobei wir für jede Sichtweise die entsprechenden Darstellungsformen in diesem Kapitel an Beispielen einführen.

	Ablauf/Fluss	Hierarchie
Programm	Programmablaufplan (PA), Nassi-Shneiderman-Diagramme (Struktogramme), Pseudo-Code	Programmhierarchiediagramm (PH)
Daten	Datenflussdiagramm (DF)	Datenhierarchiediagramm (DH)

Diese Darstellungsformen sind (mit Ausnahme des Pseudo-Codes) genormt nach DIN 66261 (Struktogramme) und DIN 66001. Will man den Programmablauf zusammen mit dem Datenfluss in einem Diagramm darstellen, so dienen dazu die Programmablaufpläne mit Daten (PAD) und Programmnetze (PN), wobei die PAD hier als Vorstufe zu den (ebenfalls genormten) PN betrachtet werden – PADs sind hingegen nicht Teil der DIN 66001.

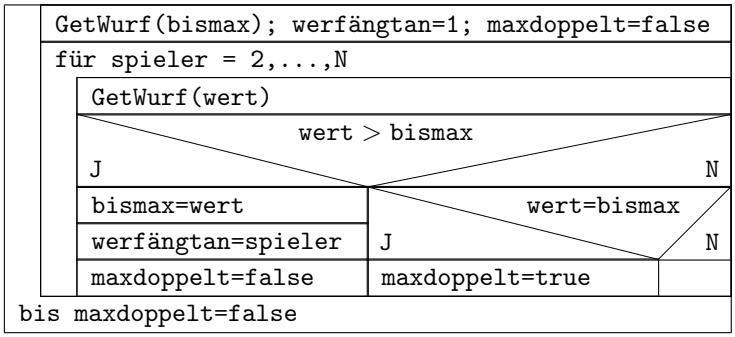
1.6.1 Erweiterte Struktogramme, das EVA-Prinzip, Datenflussdiagramme (DF) und Programmablaufpläne (PA)

Wir versuchen hier nun ein Teilproblem aus dem Mensch-Ärgere-Dich-Nicht-Spiel zu lösen: das Auswürfeln, wer beginnt. Als Eingabe haben wir die Anzahl der Spieler, Ausgabe ist die Nummer des Spielers, der im Spiel dann beginnt.

Formulierung der Aufgabe Hier zu modellierender Ablauf: jeder würfelt, der mit der höchsten Zahl ist der gesuchte Spieler. Falls mehrere die gleiche größte Zahl gewürfelt haben, wiederholen wir den Vorgang. (Übung für Fortgeschrittene: lassen Sie nur diejenigen nochmal würfeln, die die höchste Zahl gewürfelt hatten.)

Formalisierung des Ablaufs: Zunächst das Gerüst „Finde Maximum“: Merke die erste Zahl als bisher größte (Variable `bismax`), vergleiche jede weitere mit der bisher größten. Zusätzlich müssen wir uns merken, wenn die bisher größte Zahl doppelt gewürfelt wurde (Variable `maxdoppelt`). Beachte, dass bei einem neuen Maximum wir uns zum einen die entsprechende Spielernummer merken müssen (Variable `werfängtan`) und zum anderen die Variable `maxdoppelt=false` gesetzt werden muss (selbst überlegen, was sonst z. B. bei einer Wurffolge 3 2 3 5 passieren würde). Wir nehmen hier an, dass wir eine Funktion `GetWurf()` haben, die uns eine Zufallszahl zwischen 1 und 6 liefert. Die Eingabe der Anzahl der Spieler liege als Variable `N` vor.

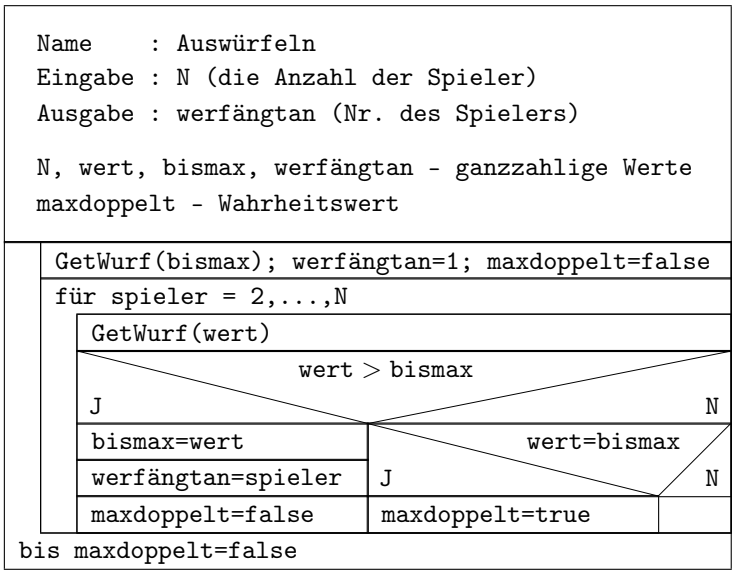
Betrachten wir dies zunächst als Struktogramm:



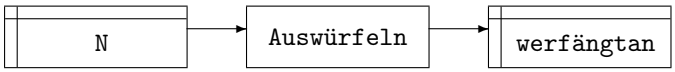
An dem Struktogramm wird nicht deutlich, was das Ergebnis der Berechnung sein soll. Ist es der Spieler `werfängtan` oder doch die höchste gewürfelte Zahl `bismax`?

Das EVA-Prinzip Die Aufgabe eines Algorithmus ist die **E**ingangsdaten einzulesen (hier das `N`), diese dann gemäß der **V**erarbeitungsvorschrift zu verarbeiten (hier das wiederholte `N`-fache Würfeln bis ein eindeutiger Spieler bestimmt ist) und schließlich die **A**usgabe (hier die Variable `werfängtan`). Die Ein- und Ausgabe (zusammen mit einem Namen) ist die Schnittstelle des Algorithmus zum aufrufenden Programm (das wäre hier das Mensch-Ärgere-Dich-Nicht-Spiel).

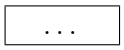
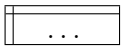
Zur Beschreibung der Schnittstelle (und der Datentypen der verwendeten Variablen) erweitert man die Struktogramme um einen Kopf. Das erweiterte Struktogramm zu unserem Beispiel ist dann:



Datenflussdiagramme (DF) sind ebenfalls zur Beschreibung der Schnittstelle geeignet. Für unser Beispiel `Auswürfeln` sieht das zugehörige Datenflussdiagramm wie folgt aus:



Die verwendeten Sinnbilder sind

-  – Befehle des Algorithmus (oder wie oben der Name des Algorithmus als abstrakter Befehl im aufrufenden Programm)
-  – Daten im Speicher des Rechners
- – Datenfluss mit Richtungsangabe

Das Datenflussdiagramm zeigt an

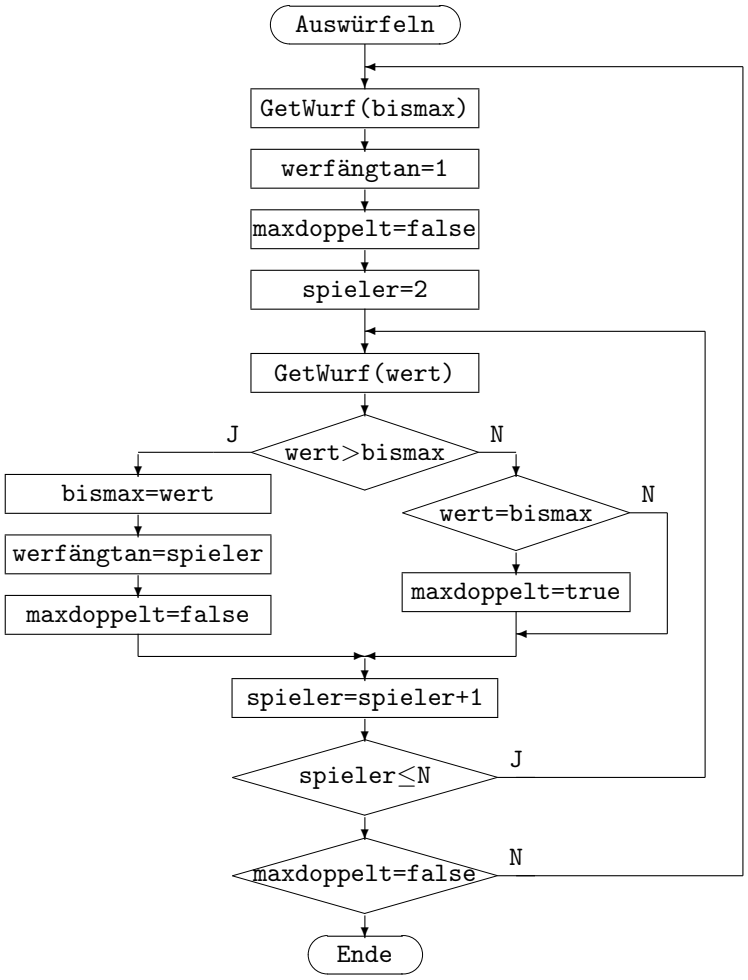
- die Richtung des Datenflusses
- die Zugriffsrechte auf die Variablen (nur lesend oder auch schreibend)
- welche Operationen auf welchen Daten ausgeführt werden

Wir benutzen Datenflussdiagramme hier zunächst nur zur Beschreibung der Schnittstelle von Algorithmen – wir kommen auf weitere Anwendungsmöglichkeiten später zurück.

Programmablaufpläne (PA) sind auch unter dem Namen Flussdiagramm bekannt. Der Ablauf wird durch mit Pfeilen verbundene Anweisungen (in Rechtecken) dargestellt. Neben dieser Reihenfolge der Operationen sieht man einem PA auch an, welche übergeordneten Operationen auf welche untergeordneten Operationen auf der nächst tieferen Abstraktionsebene

verfeinert werden (in unserem Beispiel wird der abstrakte Befehl `Auswürfeln` in mehrere Operationen aufgeteilt).

Als zunächst einziges Strukturelement dient eine Fallunterscheidung (Darstellung als Raute). Unser Algorithmus zum Auswürfeln hat dann folgende Form

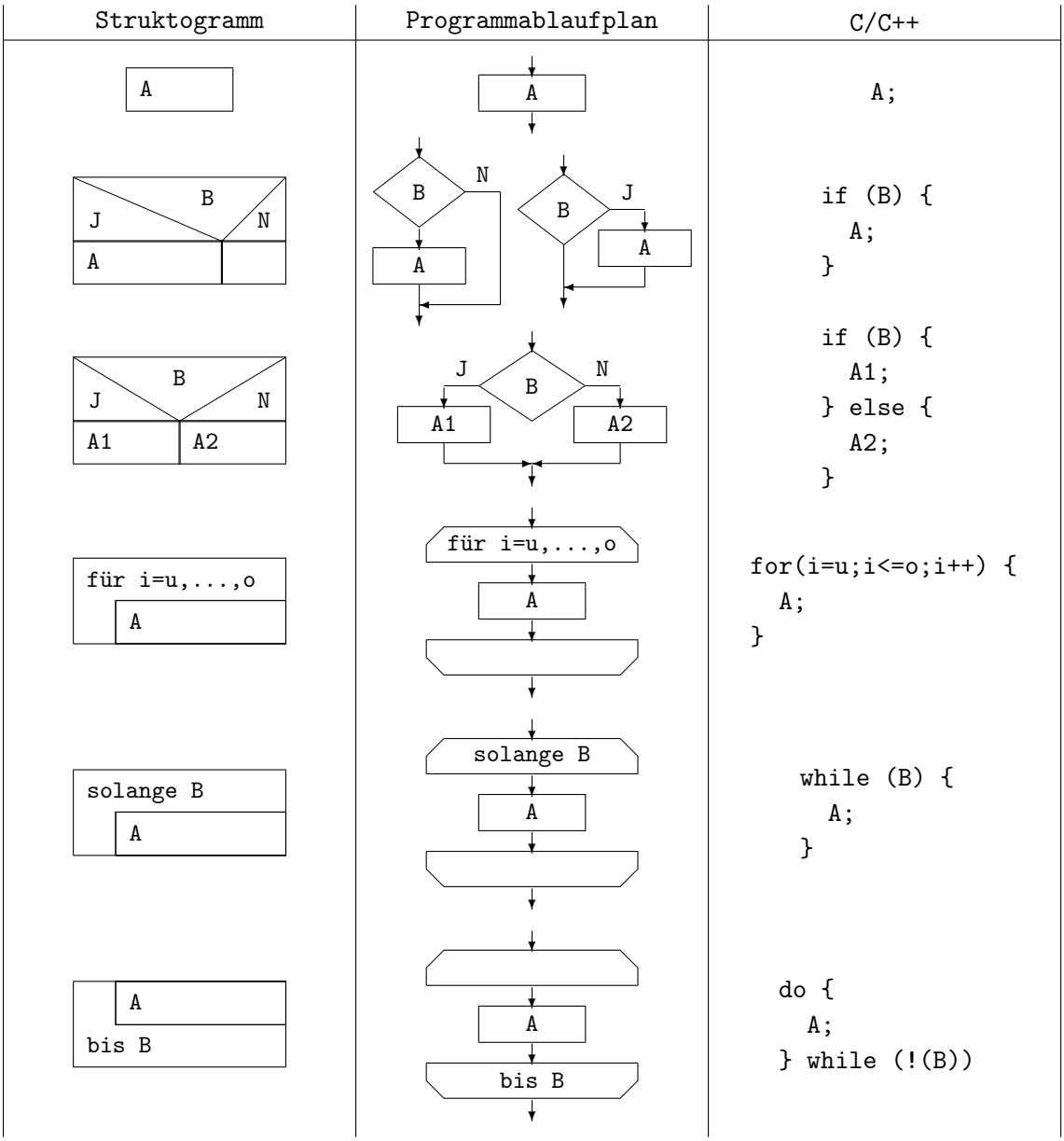


Auch wenn sich der Ablauf eines Algorithmus in solch einem PA leicht verfolgen lässt, wird man unter Umständen Schwierigkeiten haben, wenn man solch einen (unstrukturierten) Entwurf in ein C/C++-Programm umsetzen will.

1.6.2 Strukturierter Entwurf und dessen Umsetzung in C/C++

Ein Entwurf (oder Algorithmus oder Programm) heißt strukturiert, wenn er sich auf folgende Strukturelemente beschränkt: Sequenz von Anweisungen, Fallunterscheidungen und Schleifen. Bei der Umsetzung in C/C++ wollen wir hier nun eine Methode vorstellen, bei der man lediglich jedes Element des PA durch ein entsprechendes C/C++-Konstrukt ersetzen muss. Es gibt zwar stets viele Möglichkeiten, einen Entwurf in ein lauffähiges Programm umzusetzen und somit unter Umständen von dieser Methode abzuweichen, es muss dann aber stets von Hand sichergestellt werden, dass die modellierten Abläufe und Zugriffsrechte auch wirklich eingehalten werden. Die hier vorgestellte strukturierte Umsetzung garantiert dies automatisch und ist zudem leicht durchzuführen.

Wir stellen hier jeweils die Darstellung als Struktogramm, Programmablaufplan und abstrakte Umsetzung in C/C++ gegenüber:

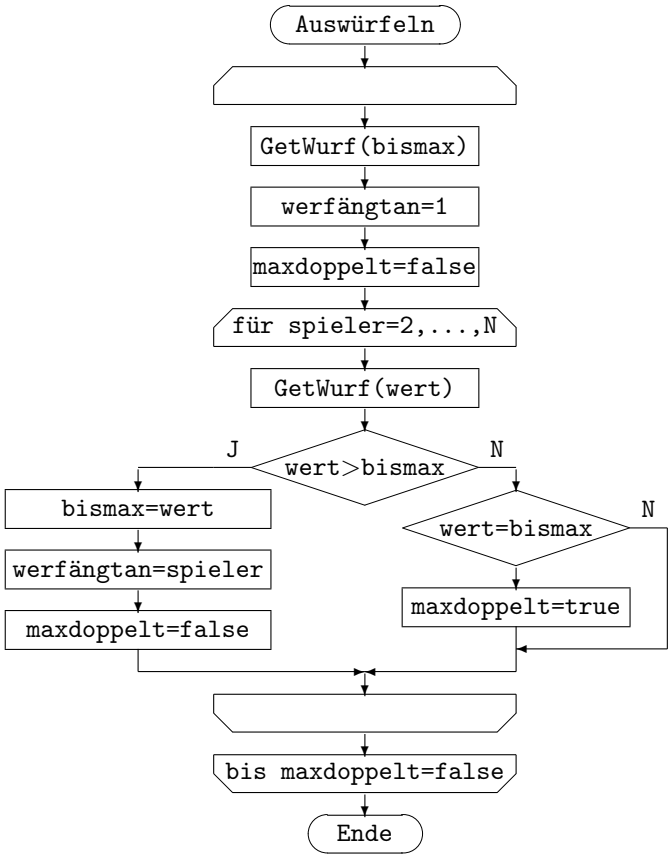


Man beachte dabei, dass in der fußgesteuerten Schleife bei der Umsetzung in C/C++ die Bedingung negiert werden muss (`do { ... } while (...)` wiederholt solange die Bedingung gilt, umzusetzen ist aber eine Schleife „wiederhole bis die Bedingung gilt“). Die Negierung wird durch das eingefügte `(!(...))` erreicht – dies hat den Vorteil gegenüber einer Negierung der Bedingung „von Hand“, dass man die Bedingung 1:1 übernehmen kann (die äußeren runden Klammern verlangt die Syntax; die inneren sind nötig, damit sich die Negierung auf die gesamte Bedingung bezieht und nicht nur auf den Beginn).

Die Negierung beliebiger boolescher Ausdrücke ist nicht ganz trivial (Stichwort „Regeln von De Morgan“, `http://de.wikipedia.org/wiki/De_Morgansche_Gesetze`). Ein Beispiel, das beim „Würfeln“ vielleicht auftreten könnte: Die Negierung von `w>=1 && w<=6` würde man entweder

leicht negieren durch `!(w>=1 && w<=6)` oder mit De Morgan umformen zu `!(w>=1) || !(w<=6)`, was sich dann noch zu `w<1 || w>6` vereinfachen ließe. Wenn mehr als zwei Terme verknüpft sind, wird die Negierung schnell unübersichtlich – wir bleiben hier bei der einfachen Variante stets den gesamten Ausdruck mit `!(...)` zu negieren.

Während man im obigen PA zum Auswürfeln die Übergänge an den „Schleifenenden“ als Sprünge im Programm interpretieren könnte, wird die Struktur erst in folgendem PA deutlich:



Auf den (strukturierten) PA angewendet, erhalten wir zunächst den folgenden C/C++-Code:

```
do {
  GetWurf(bismax);
  werfaengtan=1;
  maxdoppelt=false;
  for(spieler=2;spieler<=N;spieler++) {
    GetWurf(wert);
    if (wert>bismax) {
      bismax=wert;
      werfaengtan=spieler;
      maxdoppelt=false;
    } else {
      if (wert==bismax) {
       maxdoppelt=true;
  } } }
} while (!(maxdoppelt==false));
```

Bei der Umsetzung muss man lediglich die Feinheiten von C/C++ beachten! Bezeichner dürfen keine Umlaute enthalten. Wichtiger ist jedoch: Ein Test auf Gleichheit wird mit zwei Gleichheitszeichen `==` durchgeführt. Leider hat selbst an dieser Stelle ein Gleichheitszeichen in C/C++ eine Bedeutung – dieses würde bei einer fußgesteuerten Schleife (`do {...} while (!(maxdoppelt=false))`) dazu führen, dass die Zuweisung(!) `maxdoppelt=false` ausgeführt wird. Als Wahrheitswert würde stets der zugewiesene Wert `false` genommen, dieser aufgrund des `!(...)` noch zu `true` negiert und so würde die Schleife nie erfolgreich abgebrochen. Wenn sich Ihre Programme anscheinend unkorrekt verhalten und sonstige Fehler auszuschließen sind, schauen Sie, ob in den Bedingungen von Fallunterscheidungen oder Schleifen sich vielleicht solch ein Fehler eingeschlichen hat.

Die Umsetzung in C/C++ ist so noch nicht fertig. Neben der Deklaration der Variablen (diese lassen sich leicht oben einfügen) wurden insbesondere die im Datenflussdiagramm modellierten Zugriffsrechte nicht beachtet. Wir könnten – ohne dass sich der Compiler beschwert – der Variablen `N` einen neuen Wert zuweisen, also schreibend auf sie zugreifen, obwohl nach DF nur lesender Zugriff erlaubt war. Zur automatischen Berücksichtigung werden wir bei der Umsetzung in C/C++-Code die einzelnen Algorithmen kapseln und Funktionen schreiben, deren Ein- und Ausgabeparameter genau die durch das DF definierte Schnittstelle realisieren – inklusive der modellierten Zugriffsrechte. Dabei werden die Eingangsvariablen mit ausschließlicher Leseberechtigung als `const` Variablen deklariert, die Ausgangsvariablen (Schreibberechtigung nötig) mit einem `&` versehen (wir gehen im nächsten Abschnitt über Parameterübergabe-Mechanismen auf die genauere Bedeutung des `&` ein).

Bei der Umsetzung der Diagramme mit Hilfe der hier vorgestellten Methode werden sämtliche Ergebnisse von Funktionen über Referenzparameter (dies sind die Parameter mit dem `&`) an das aufrufende Programm zurückgegeben. Das Rückgabekonzept von C/C++ bei Funktionen wird hier nicht verwendet – allen Funktionsnamen wird somit das Wort `void` vorangestellt (bis auf das vom Sprachstandard vorgeschriebene `int` bei `int main(...)` – dass Microsoft den Standard ignoriert und auch `void main(...)` zulässt, ist ein anderes Thema – belassen Sie es bei `int main(...)`, insbesondere, wenn Ihre Programme auch mit anderen C/C++-Compilern übersetzbar bleiben sollen). Wir werden auf das Rückgabekonzept von C/C++ an späterer Stelle zurückkommen.

Zur Verwendung der Funktion müssen wir noch das zugehörige aufrufende Programm hinzufügen. Hier nun das entsprechende komplette Programm. Dieses demonstriert nebenbei noch die Verwendung des Zufallszahlengenerators in C/C++ (wir gehen auf dessen Details nicht weiter ein).

```
#include <iostream>  // für die einfache Textausgabe
#include <ctime>     // für time() zur Initialis. des Zufallszahlengenerators
#include <cstdlib>   // für srand() und rand() für Zufallszahlen

using namespace std; // für iostream

void GetWurf(int & wurf) {
  wurf = rand() % 6 +1; // rand() liefert eine ganzzahlige Zufallszahl,
  // das % rechnet diese modulo 6 (Werte zwischen 0 und 5, deshalb noch +1)
}
```

```
void auswuerfeln(const int N, int & werfaengtan) { // Parameter & Datentypen
  // der Parameter N erlaubt nur Lesezugriff (const),
  // der Parameter werfaengtan erlaubt Schreibzugriff (&)
  int bismax, spieler, wert; // fehlende lokale Variablen deklarieren
  bool maxdoppelt;
  do {
    GetWurf(bismax);
    werfaengtan=1;
    maxdoppelt=false;
    for(spieler=2;spieler<=N;spieler++) {
      GetWurf(wert);
      if (wert>bismax) {
        bismax=wert;
        werfaengtan=spieler;
        maxdoppelt=false;
      } else {
        if (wert==bismax) {
          maxdoppelt=true;
        }
      }
    }
  } while (!(maxdoppelt==false));
}

int main(void) {
  // einmal(!) im Hauptprogramm für die Initialisierung
  srand(unsigned(time(0))); // des Zufallszahlengenerators aufrufen
  // ab hier Variablen deklarieren
  const int spieleranzahl=4;
  int derfaengtan;
  // Aufruf und Ausgabe
  auswuerfeln(spieleranzahl,derfaengtan);
  cout << "Der Spieler " << derfaengtan << " fängt an!" << endl; //
  // zum Abschluss noch das fehlerfreie Beenden an die Konsole melden
  return 0;
}
```

Man beachte, dass es keine Variablen-Deklarationen außerhalb von `main` gibt! Das heißt insbesondere, dass die von uns geschriebenen Funktionen ausschließlich auf die formalen Parameter und lokalen Variablen zugreifen können. Unbeabsichtigtes Ändern von Variablen aus dem Hauptprogramm sind somit ausgeschlossen. Dies ist ein nicht zu unterschätzender Vorteil!

Durch die restriktive Umsetzung aller Zugriffsrechte und Ausschluss von globalen Variablen ermöglichen wir dem Compiler, möglichst viele Tippfehler oder Unachtsamkeiten als formale Fehler erkennen zu können.

1.6.3 Parameterübergabe-Mechanismen und ein einfacher Sortieralgorithmus

C/C++ stellt die klassischen Parameterübergabe-Mechanismen „call-by-value“ und „call-by-reference“ zur Verfügung. Grundsätzlich müssen die Datentypen der formalen Parameter (dies sind die in der Deklaration der Funktion) und die der aktuellen Parameter (dies sind die des konkreten Aufrufs einer Funktion) übereinstimmen.

Die Funktionsweise: Bei „call-by-value“ werden die Werte der aktuellen Parameter kopiert und den formalen Parametern zugewiesen. Die Funktion greift danach ausschließlich auf die formalen Parameter zu und hat keine Möglichkeit festzustellen, woher die Werte beim Aufruf kamen. Die aktuellen Parameter können Literale (direkt angegebene Werte), Variablen, aber auch beliebige Terme sein (sofern sich diese zu dem Datentyp des zugehörigen formalen Parameters auswerten).

Bei „call-by-reference“ wird statt einem Wert die Adresse einer Speicherstelle übergeben. Dies hat zur Folge, dass jeder Zugriff auf einen formalen Parameter auf die Speicheradresse des zugehörigen aktuellen Parameters zugreift und so der formale und aktuelle Parameter stets denselben Wert haben. Die aktuellen Parameter können bei „call-by-reference“ grundsätzlich nur Variablen sein (selbst überlegen warum).

In anderen Programmiersprachen gibt es zum Teil noch weitere Möglichkeiten, die in C/C++ nicht realisiert sind. Eine Übersicht bietet `http://en.wikipedia.org/wiki/Evaluation_strategy`.

Die Umsetzung: „call-by-value“ ist bei C/C++ die Voreinstellung (die formalen Parameter werden normal als Datentyp und Variablenname angegeben). Bei der Umsetzung der strukturierten Entwürfe verwenden wir „call-by-value“ für die Eingangsparameter (sofern diese nicht gleichzeitig Ausgangsparameter sind). Dabei deklarieren wir die zugehörigen formalen Parameter als Konstanten (Angabe des Schlüsselwortes `const` vor dem Datentyp) um die Zugriffsrechte aus dem Datenflussdiagramm explizit zu berücksichtigen.

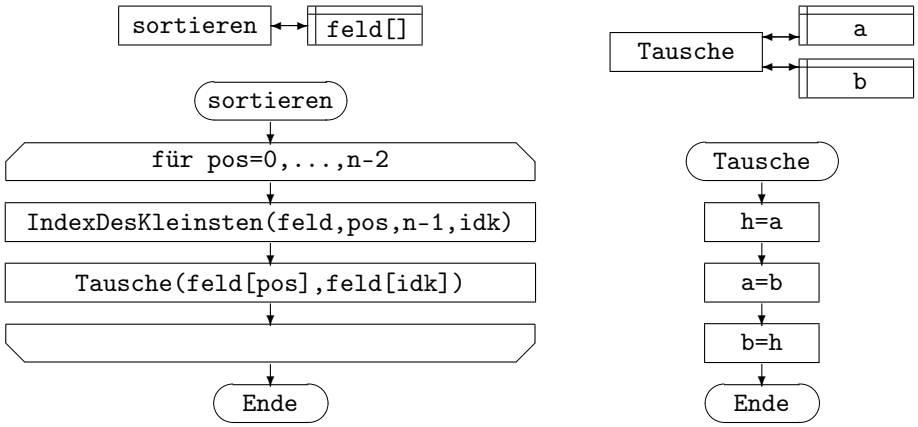
In C++ wird „call-by-reference“ über sogenannte Referenzparameter umgesetzt. Ein formaler Parameter wird als Referenzparameter durch ein `&` zwischen Datentyp und Variablenname gekennzeichnet. Innerhalb der Funktion kann dieser formale Parameter wie eine normale Variable verwendet werden. Wir benutzen dieses Konzept bei Ausgangsvariablen (also wenn das Datenflussdiagramm eine notwendige Schreibberechtigung anzeigt).

Referenzparameter sind ein Teil der Sprache C++, die in dieser Form in C nicht zur Verfügung stehen. Will man in C „call-by-reference“ umsetzen, so geht dies nur über das Konzept der Zeiger (Pointer).

Wir demonstrieren die Unterschiede an folgendem Übungsbeispiel:

Ein Aspekt des Auswürfelns war die Bestimmung des Maximums einer Menge von Zahlen. Wir entwerfen basierend auf dieser Idee einen Sortieralgorithmus: Gegeben sei ein Array mit Zahlen – finde die kleinste Zahl und tausche diese an die erste Position; finde die nächstkleinste Zahl im Feld (Suche ab der zweiten Position im Feld) und tausche diese an die zweite Position; wiederhole dieses Prinzip bis zur letzten Position. Das Verfahren heißt Sortieren durch Minimumsuche (oder kurz „Minimumsort“).

Der Programmablaufplan für die Sortier-Funktion und die darin enthaltene Tausch-Funktion, sowie die zugehörigen Datenflussdiagramme werden hier angegeben (die Funktion zum Finden der Position des kleinsten Elements bleibt hier zur Übung offen).



An einem Teil der Lösung – dem Tausch zweier Elemente – betrachten wir die Unterschiede bei der Umsetzung des call-by-reference-Parameterübergabe-Mechanismus. Mit Referenzparametern (C++-Style) erhalten wir folgende Funktion:

```
void Tausche(int & a, int & b){
  int h;
  h = a;
  a = b;
  b = h;
}
```

Der Aufruf innerhalb des Sortieralgorithmus lautet wie im PA angegeben `Tausche(feld[pos],feld[idk]);` – die Verwendung der Variablen innerhalb der Tausche-Funktion und bei deren Aufruf unterscheidet sich in keiner Weise von der anderer Variablen. Beschränkt man sich auf reines C, so bleibt nur die Verwendung von Zeigern – die Funktion sähe dann so aus:

```
void Tausche(int * a, int * b){
  int h;
  h = *a; // der Variablen h wird der Wert zugewiesen,
  // der in der Speicherstelle steht, die durch die Adresse
  // referenziert ist, die in der Variablen a steht
  *a = *b; // dito bzgl. b, aber der Wert wird nicht der Variablen
  // direkt zugewiesen, sondern an die Speicherstelle geschrieben,
  // die in der Variablen a steht
  *b = h; // der Wert wird direkt aus h ausgelesen und an die
  // Speicherstelle geschrieben, die in der Variablen b steht
}
```

Der Aufruf müsste dann – anders als oben – `Tausche(&feld[pos],&feld[idk]);` lauten (über den Adressoperator `&` wird in `&feld[pos]` die Adresse, an der der Wert `feld[pos]` steht, ermittelt und diese als Parameter übergeben). Die Verwendung von Zeigern führt leicht zu schwer zu findenen Fehlern – man muss sich ständig bewusst sein, ob man auf den Inhalt der Variablen selbst zugreift oder ob die Variable nur eine Speicheradresse angibt, an der der eigentliche Wert steht. Beim Aufruf muss (wie oben geschrieben) über den Adressoperator `&`

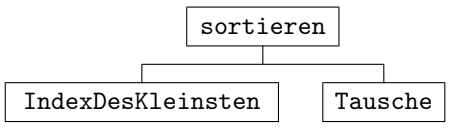
dem Compiler explizit mitgeteilt werden, dass ein „call-by-reference“ gewünscht wird. (Man mag dieses vielleicht als einzigen(?) kleinen Vorteil gegenüber dem Referenzparameterkonzept erachten, dass man am Aufruf erkennen kann, dass ein „call-by-reference“ vorliegt – dass man also damit rechnen muss, dass sich der Wert der Variablen durch den Aufruf verändert.)

Fazit: Das Referenzparameterkonzept von C++ ist weit weniger fehleranfällig, so dass wir uns bei der Umsetzung der Entwürfe auf diese Methode beschränken.

Randbemerkung (Blick für Fortgeschrittene hinter die Kulissen zum tieferen Verständnis): Der Referenzoperator `&` in C++ und der Adressoperator `&` in C/C++ sind zwar verwandt, aber in der Verwendung trotzdem grundsätzlich verschieden! Der Referenzoperator `&` verleiht einer Variablen einen zweiten Namen, mit dem ebenfalls auf diese zugegriffen werden kann – man kann sich solche Referenzen prinzipiell auch wie normale Variablen anlegen (nach `int a; int & b = a;` hätte man zwei Variablen `a` und `b`, die auf dieselbe Speicheradresse zugreifen), eine spätere Änderung des Referenzoperators auf eine andere Variable ist nicht möglich. Der Referenzoperator kann ausschließlich(!) bei der Deklaration und bei formalen Parametern verwendet werden. Der Adressoperator hingegen wird an allen Stellen benötigt, an denen eine Adresse einer Variablen ermittelt werden muss. Zeigervariablen in C/C++ lassen sich nach der Deklaration beliebig auf andere Speicheradressen umbiegen. Nach `int a; int * b = &a;` könnte man auf die Variable `a` auch über `*b` zugreifen, ein späteres Ändern des Zeigers `b` auf eine andere Variable (z. B. `int c; b= &c;`) ist problemlos möglich[1]. Referenzparameter kann man somit am ehesten noch mit einer Deklaration eines konstanten Zeigers vergleichen: `int a; int * const b = &a;` (hierbei kann die Speicherstelle, auf die `b` verweist, nachträglich nicht mehr verändert werden – der Wert, der dort steht, hingegen schon; soll stattdessen der Wert der referenzierten Speicherstelle unverändert bleiben, müsste die Deklaration `const int * b;` (oder `int const * b;`) lauten – dann dürfte man über `b` zwar beliebige Speicherstellen referenzieren, auf diese aber stets nur lesend zugreifen).[2] Wir kommen auf diese Feinheiten (in vereinfachter Form) im Rahmen der zusammengesetzten Datentypen (Stichwort `struct`) nochmals zurück.

1.6.4 Das Programmhierarchiediagramm (PH)

Durch das Aufteilen der Funktion `sortiere` in die Teilfunktionen `IndexDesKleinsten` und `Tausche` haben wir implizit eine Hierarchie der nun drei Funktionen eingeführt: `sortiere` verwendet die beiden anderen, aber nicht umgekehrt. Diese Aufrufbeziehungen stellen wir in einem Programmhierarchiediagramm (PH) dar. Es zeigt neben den Aufrufbeziehungen an, welche übergeordneten Operationen in welche untergeordneten Operationen (der nächst tieferen Abstraktionsebene) verfeinert wurden. Darüberhinaus gibt es bei umfangreicheren Projekten einen Überblick, wie die verschiedenen Funktionen miteinander interagieren. Dargestellt werden Funktionen in Rechtecken und die Hierarchie als Linien (in der Regel ohne Pfeilspitzen) von oben nach unten.



[1] Zur Illustration der Verwechslungsgefahr noch ein Beispiel: in `b=&c;` wird dem Zeiger `b` die Adresse der Variablen `c` zugewiesen, `*b=c;` hätte hingegen an die Adresse, die der Zeiger `b` referenziert, den Wert der Variablen `c` geschrieben.

[2] Als Gedächtnisstütze kann hierbei die Regel „lese von rechts nach links“ helfen: `int * const` ist dann ein konstanter Pointer auf eine Ganzzahl, `const int *` ein Pointer auf eine Ganzzahl-Konstante.

Die Verbindungen lassen sich von oben nach unten als „... verwendet ...“ lesen; im Beispiel also „die Funktion `sortieren` verwendet die Funktion `IndexDesKleinsten`“ und „die Funktion `sortieren` verwendet die Funktion `Tausche`“. Ein umfangreicheres Beispiel für ein Programmhierarchiediagramm folgt noch in Abschnitt 1.6.8, wenn wir alle Module und Funktionen des Mensch-Ärgere-Dich-Nicht-Projektes zusammenfassen.

1.6.5 Lebensdauer, Gültigkeit und Sichtbarkeit von Variablen

Haben lokale Variablen (die formalen Parameter und andere in einer Funktion deklarierte Variablen) und globale Variablen dieselben Bezeichner, muss man sich Gedanken machen, auf welche konkrete Variable man mit dem Bezeichner dann zugreift. Dieses führt auf die Begriffe Lebensdauer und Gültigkeit / Sichtbarkeit von Variablen.

Unter dem Sichtbarkeitsbereich einer Variablen versteht man den Programmabschnitt, in dem die Variable nutzbar und sichtbar (also gültig) ist. Da eine lokale Variable denselben Namen tragen darf wie eine globale Variable, sind Sichtbarkeitsbereiche nicht notwendig zusammenhängend: durch die Deklaration einer lokalen Variable wird die gleichnamige globale Variable für einen bestimmten Block „verschattet“, d. h., sie ist temporär nicht sichtbar.

Unter der Lebensdauer einer Variablen versteht man den Zeitraum, in dem für die Variable Speicherplatz reserviert ist. Wird der Speicherplatz für andere Zwecke wieder freigegeben, so „stirbt“ die Variable und ist nicht mehr nutzbar. Lokale Variablen werden bei jedem Aufruf der Funktion erstellt; in der Regel wird der Speicherplatz beim Verlassen der Funktion wieder freigegeben.

In C/C++ gibt es einen Zusatz `static` bei der Deklaration, der nur die Sichtbarkeit einer Variablen auf den Namensraum der Funktion einschränkt, nicht aber ihre Lebensdauer. Die Sichtbarkeit einer solchen Variablen verhält sich also wie die einer lokalen Variablen, die Lebensdauer dagegen wie die einer globalen, d. h., beim Eintritt in die sie umschließende Funktion hat sie exakt den selben Wert wie am Ende des letzten Aufrufs der Funktion. Bei strukturierten Entwürfen können solche als `static` deklarierten Variablen nicht vorkommen.

Bei unseren bisherigen Anwendungen stimmt das Verhalten der Variablen mit dem mit gesundem Menschenverstand ermittelten überein. Wir werden erst bei Rekursion (Abschnitt 1.7) noch einmal genauer auf diese Aspekte der Programmierung eingehen müssen.

1.6.6 Eigene Datentypen mit `typedef` und `struct` und das Datenhierarchiediagramm (DH)

Wir haben bereits die einfachen Datentypen für ganze Zahlen (`int`, `signed`/`unsigned`/`short`/`long`/`long long int`), Gleitkommazahlen (`float`, `double`, `long double`), Zeichen (`char`) und Wahrheitswerte (`bool`) kennengelernt. Ebenso Arrays als Sammlung gleicher Datentypen unter einem Variablennamen (z. B. ein Feld `a` mit 50 `int`-Werten: `int a[50];`). In diesem Abschnitt wollen wir zunächst die Möglichkeiten für weitere, selbstdefinierte Datentypen skizzieren und danach eine Darstellungsform für den Entwurf vorstellen, der – wie bei DF und PA – eine einfache Umsetzung in C/C++-Code erlaubt.

Umbenennung von Datentypen ist die einfachste (aber auch eingeschränkteste) Möglichkeit neue Datentypen in C/C++ anzulegen. Die Syntax lautet

```
typedef <gültiger Datentyp> <neuer Name>;
```

Diese reine Umbenennung von Datentypen hat (in C/C++) keinerlei Auswirkungen auf das Verhalten der Variablen im Programm (für den C/C++-Compiler ist es egal, ob eine Variable als `int a;` oder nach einem `typedef int Ganzzahl;` als `Ganzzahl a;` deklariert wird). Trotzdem kann die Anwendung des `typedef`-Befehls sinnvoll sein – zwei gute Gründe sind:

- Der Datentyp für bestimmte Variablen könnte später geändert werden müssen.

  Erinnern wir uns an das Beispiel der Wettervorhersage: die Temperaturdaten könnten in einem ersten Entwurf als `int`-Werte vorgesehen gewesen sein – das Programm wurde geschrieben; es sind mehrere tausend Zeilen C/C++-Code entstanden; nun stellt man fest, dass bei der Weiterverarbeitung der Temperaturwerte als Ganzzahlen es zu zu großen Ungenauigkeiten kommt und die Temperaturvariablen als `float` hätten deklariert werden müssen. Es werden sicher nicht alle `int`-Variablen nun zu `float` umgewandelt werden müssen – man muss dieses also von Hand machen – bei mehreren tausend Zeilen Code sind dabei Fehler kaum zu vermeiden.

  Hätte man stattdessen ein einfaches

  ```
  typedef int Temperatur;
  ```

  eingefügt und konsequent alle Variablen für Temperaturwerte mit dem Datentyp `Temperatur` deklariert, so würde die Änderung aller Temperatur-Variablen auf `float` fehlerfrei durch Änderung nur der einen `typedef`-Zeile möglich sein.

- Auch, wenn so eine nachträgliche Änderung ausgeschlossen scheint, können solche eigenen Datentypen die Lesbarkeit von Programmen erhöhen.

  Die Intention einer Variable lässt sich unter Umständen schneller erschließen, wenn man dem Programmcode entnehmen kann, dass es sich um eine Variable z. B. für Temperaturen handelt – wenn als Datentyp hingegen nur `int` oder `float` angegeben ist, bedarf es mehr Aufwand, die entsprechende Funktion zu verstehen.

Verbunde (auch Records genannt): Während bei Arrays auf eine Vielzahl gleicher Datentypen über einen Variablennamen und einen Index zugegriffen wird, bieten Verbunde die Möglichkeit verschiedene Datentypen unter einem Namen zusammenzufassen – das zugehörige C/C++-Konstrukt heißt `struct`. Bei `struct` gibt es einige feine Unterschiede zwischen C und C++ – die Informationen, die hierzu im Internet (speziell in Foren) zu finden sind, widersprechen sich leider oft, was wohl im Kern darauf zurückzuführen ist, dass sich viele Benutzer dieser Unterschiede nicht bewusst sind. Wir stellen hier zunächst die C++-Variante vor und gehen auf die Unterschiede in C danach ein. Die Syntax zur Deklaration eines `struct`-Datentyps (in C++) lautet:

```
struct <TypName> {
  <Datentyp_1> <Komponente_1>;
  <Datentyp_2> <Komponente_2>;
       ⋮              ⋮
  <Datentyp_N> <Komponente_N>;
} ;
```

Eine Komponente eines Verbundes nennt man Selektor. Eine Variable des Structtyps kann dann (wie bei jeder anderen Variable auch) durch

```
<TypName> <VarName>;
```

deklariert werden. Ein Beispiel: Arrays stellen eine feste Anzahl von Elementen zur Verfügung – wollen wir eine variable Anzahl von Elementen verwalten, so könnten wir ein ausreichend großes Array zur Verfügung stellen und uns in einer zweiten Variablen die aktuell tatsächlich verwendete Anzahl merken. Die maximale Anzahl der Elemente würde man sich in einer geeigneten Konstanten definieren.

```
const int MAX_ANZ = 100;
typedef int ElementTyp;
typedef ElementTyp Datenbehaelter[MAX_ANZ];
// die Syntax von typedef ist manchmal verwirrend:
// obiges definiert einen Typ Datenbehaelter, der ein Array ist,
// das aus MAX_ANZ Variablen des Typs ElementTyp besteht
struct Menge {
 Datenbehaelter inhalt; // inhalt ist jetzt ein Array mit MAX_ANZ Elementen
 int anzE; // Anzahl der Elemente => im Array werden die Indizes
 // 0 bis anzE-1 verwendet, anzE ist also gleichzeitig der erste freie Index
 // (außer anzE == MAX_ANZ, dann ist die Kapazität des Arrays erschöpft)
} ;

Menge m; // eine Variable des obigen struct-Typs
m.anzE = 0; // zu Beginn sind keine Einträge vorhanden -> Menge ist leer
m.inhalt[0]=42;
m.anzE++; // m ist jetzt eine Menge mit der 42 als einzigem Element
```

Bei der Typdefinition können die einzelnen Komponenten leider nicht mit Default-Werten vorbelegt werden, so dass jede Variable des Structtyps zu Beginn einen definierten Wert hätte – man muss dieses nach der Deklaration der Variablen stets selbst durchführen (hier `m.anzE=0;`). Der Zugriff auf die einzelnen Komponenten erfolgt über die Punkt-Notation (ein Punkt zwischen dem Namen der Structvariablen und dem Namen der Komponente, auf die zugegriffen werden soll). Verbunde können auch geschachtelt sein (ein Verbund kann einen anderen Verbund als Komponente enthalten – selbst ausprobieren).

Übungsaufgabe: Überlegen Sie sich Funktionen, die auf obigem Datentyp `Menge` arbeiten (z. B. Hinzufügen eines Elements, Finden eines Elements, Test auf Leerheit, Vereinigung zweier Mengen, ...).

In C ist das Arbeiten mit Structs etwas komplizierter. Während bei C++ mit der Deklaration eines Structs der Datentyp automatisch zur Verfügung steht, muss man in C zur Deklaration einer Variablen dem Compiler nochmal explizit mitteilen, dass eine Variable eines Structs angelegt werden soll (im obigen Beispiel hieße es dann `struct Menge m;`) – um `Menge` direkt als Datentyp verwenden zu können, kann man das Struct als Datentyp über `typedef struct Menge Menge;` bekannt machen. Beim Versuch obiges mit einem C-Compiler zu übersetzen stellt man noch fest, dass C bei der Verwendung von Konstanten bei der Größenangabe von Arrays nicht mitspielt (trotz `const` wird es als Variable betrachtet, Arraygrößen müssen in C aber zur Kompilierzeit bekannt sein). Man kann sich hier helfen, indem man die Konstante über die Präprozessoranweisung definiert (`#define MAX_ANZ 100` – wie bei Präprozessoranweisung üblich ohne abschließendes „;“).

Man sieht auch häufiger die Form

```
typedef struct {
  <Datentyp_1> <Komponente_1>;
  <Datentyp_2> <Komponente_2>;
       ⋮              ⋮
  <Datentyp_N> <Komponente_N>;
} <TypName>;
```

Dies entspricht dem Umbenennen des Structs (ein „gültiger Datentyp“) in einen „neuen Namen“ `<TypName>` – ist also entgegen gängiger Behauptung keine weitere mögliche Syntax, sondern lediglich eine Anwendung des Befehls `typedef`. Diese Variante hat aber noch einen gewissen Charme, da sie in C und C++ gleichermaßen funktioniert.

Sie ist aber fehleranfällig. Ohne das `typedef` wird die Bedeutung plötzlich eine komplett andere!

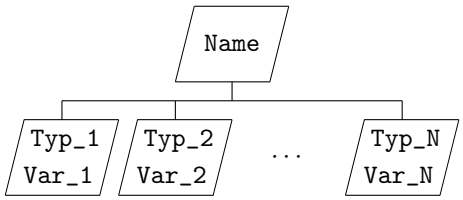
```
struct {
  <Datentyp_1> <Komponente_1>;
  <Datentyp_2> <Komponente_2>;
       ⋮              ⋮
  <Datentyp_N> <Komponente_N>;
} <Name>;
```

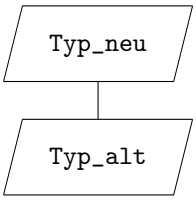
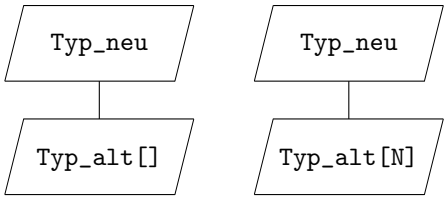
Es wird kein Typ deklariert, sondern lediglich eine Variable(!) `<Name>` angelegt mit der Struktur des Structs – es entspricht (wie ein `int a;`) der normalen Deklaration einer Variablen (`<gültiger Datentyp> <VarName>;`). Der Typ dieser Variable hätte in diesem Fall jedoch keinen eigenen Namen; Variablen vom selben Typ könnten nur durch Wiederholung des gesamten Structs deklariert werden.

Diesem fehleranfälligen Wirrwarr können Sie beim Entwurf neuer Programme leicht entgehen, wenn Sie sich an die einfachere C++-Variante halten. Bei der Wartung alter C-Programme werden Sie sich jedoch früher oder später mit den älteren Varianten herumschlagen müssen.

Darstellung als Datenhierarchiediagramm (DH): Wir stellen in einem DH die Hierarchierelationen „Datentyp A enthält die Datentypen B_1,...,B_k“ sowie „Datentyp A ist ein Datentyp B“ und „Datentyp A ist ein Array aus Elementen vom Datentyp B“ dar. Jedes (korrekte) DH setzt sich aus exakt diesen Komponenten zusammen. Die folgende Übersicht stellt die Teildiagramme und die jeweilige Umsetzung in C++ gegenüber (falls eine Umsetzung in C nötig sein sollte, beachten Sie obige Hinweise):

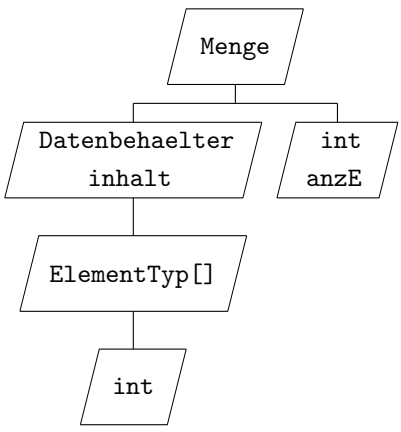
Datenhierarchiediagramm	C++

	`struct Name {`
	`  Typ_1 Var_1;`
	`  Typ_2 Var_2;`
	`  ⋮  ⋮`
	`  Typ_N Var_N;`
	`};`

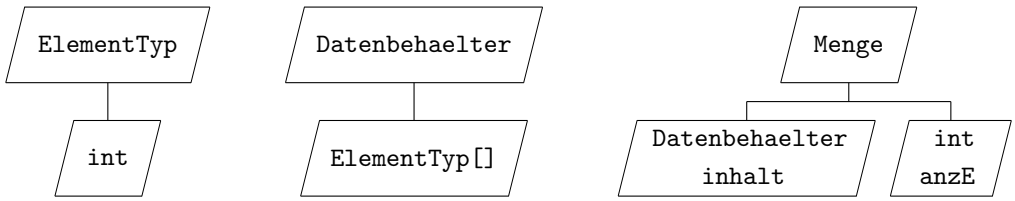
	`typedef Typ_alt Typ_neu;`
	`const int N=...;` `typedef Typ_alt Typ_neu[N];`

Bei der Umsetzung muss man beachten, dass das Datenhierarchiediagramm von „unten nach oben“ umgesetzt wird, da der Compiler sonst die verwendeten Datentypen noch nicht kennt. Fehlt bei Arrays im Datenhierarchiediagramm die Größenangabe, definieren wir uns geeignete Konstanten, um bei späteren Änderungen nur an dieser einen Stelle ändern zu müssen (Konstanten sollten in Programmen nie(!) als konkrete Werte fest kodiert werden – bei späteren Änderungen sind insbesondere in größeren Programmen sonst Fehler durch zuwenig oder zuviel geänderte Stellen im Source-Code fast unvermeidbar). Ist die Konstante `N` schon an anderer Stelle definiert, entfällt die Zeile.

Das oben genannte Beispiel des Verbunds für eine Menge ist ein einfaches Beispiel, das die drei genannten Strukturen enthält:



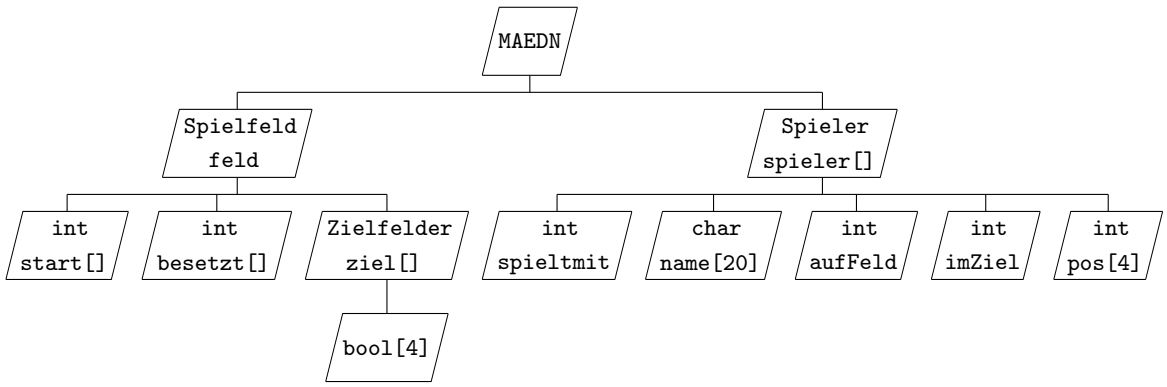
Wir sehen in diesem Beispiel auch, dass sich die drei Varianten überlappen können! Zur Verdeutlichung hier nochmal die Teildiagramme ohne Überlappung einzeln – in der Praxis werden die Einzelteile jedoch nicht angegeben sondern nur das gesamte Datenhierarchiediagramm. Bei der Umsetzung muss man die Einzelteile selbst erkennen (dies ist stets eindeutig möglich – es sei denn, das Diagramm ist fehlerhaft).



Bei der Umsetzung von unten nach oben würde hier zunächst die Umbenennung des `int` durchgeführt (`typedef int ElementTyp;` – unten steht ein Typ (`int`, also kein Array), es kann also nur die zweite Variante in obiger Übersicht sein („Typ A ist ein Typ B“) – die Arrayklammern gehören demnach zum darüberliegenden Teildiagramm). Das nächste Teildiagramm: Im oberen Kästchen eines Teildiagramms steht stets nur ein Typ (`inhalt` ist also ein Selektor des wiederum darüberliegenden Teildiagramms) – es liegt hier also ein „Typ A ist ein Array aus Elementen vom Typ B“ vor. Da die Arraygröße nicht angegeben ist, definieren wir uns eine geeignete Konstante (u. U. ist dieser Wert an anderer Stelle des Entwurfs beschrieben) ⇝ `const int MAX_ANZ = 100;` und `typedef ElementTyp Datenbehaelter[MAX_ANZ];`. Es verbleibt der Verbund, den wir ebenfalls nach obigen Schema umsetzen. Insgesamt führt dies exakt zu dem oben bereits angegebenen C++-Code.

Diese Deklarationen fügen wir stets ganz oben im Programm ein – spätestens jedoch vor den Funktionen, die diese Datentypen verwenden.

Als etwas komplizierteres Beispiel betrachten wir das Datenhierarchiediagramm, wie es bei einem Entwurf für das Mensch-Ärgere-Dich-Nicht-Spiel entstehen könnte:



In dem Verbund für das gesamte Spiel ist in der linken Hälfte das Spiel aus Sicht des Spielfelds modelliert, in der rechten Hälfte aus Sicht der Spieler). Häufig ist es effizienter mehrere Sichtweisen parallel zu verwalten als – mit weniger Speicherverbrauch – sich auf eine Sichtweise zu beschränken und dann nur mit erhöhtem Aufwand andere Informationen aus der Datenstruktur auszulesen. Angenommen, Sie hätten sich auf die linke Hälfte des DH beschränkt: für die Ausgabe des Spielfeldes wäre dies kein Problem, wollen Sie hingegen entscheiden, welche Figuren z. B. Spieler 1 ziehen kann, müssten Sie zunächst alle Felder des Spielfelds abfragen, ob dort jeweils eine Figur vom Spieler 1 steht – andersherum: hätte man nur die Spielersicht,

wäre die Ausgabe des Spielfeldes sehr kompliziert, da die Information, ob (und wenn ja mit welcher Figur) ein Feld besetzt ist, auf viele Stellen verteilt ist. Nichtsdestotrotz ist obiges nur ein Vorschlag; es gibt fast immer mehrere sinnvolle Varianten der Modellierung.

Überprüfen Sie nun, ob nach obigen Regeln wirklich der folgende C++-Code entsteht (alle Konstanten sind hier zuerst definiert), und versuchen Sie die Bedeutung der einzelnen Datenkomponenten zu erfassen:

```
const int MAX_Spieler=4;
const int MAX_Felder=10*MAX_Spieler;
typedef bool Zielfelder[4]; // Typ für den Zielbereich
struct Spielfeld {
  int start[MAX_Spieler];       // Anzahl Figuren, die nicht draußen sind
  int besetzt[MAX_Felder];      // Runde ab 1. Spieler gezählt, Wert=spielernr
  Zielfelder ziel[MAX_Spieler]; // Zielbereiche der einzelnen Spieler
};
struct Spieler {
  int spieltmit;     // 0=leer, 1=Mensch, 2=Computer
  char name[20];     // Name
  int aufFeld;       // Anzahl Figuren auf dem Feld
  int imZiel;        // Anzahl Figuren im Zielbereich
  int pos[4];        // Positionen seiner Figuren bzgl. seinem Startfeld
};
struct MAEDN {
  Spielfeld feld;
  Spieler spieler[MAX_Spieler];
};
```

Parameterübergabe bei `structs`: Nach dem bisherigen Verfahren der Umsetzung von Programmablaufplänen in C/C++-Code würden wir Eingangsparameter stets als „call-by-value“ implementieren. Für Verbunde hieße das, dass die gesamte Datenstruktur kopiert werden muss – dies ist unter Umständen sehr zeit- und speicherplatzaufwendig. Ziel des „call-by-value“ war, die ausschließliche Leseberechtigung (die im DF modelliert war) umzusetzen; daher wurde nur der Wert der Variablen übergeben, die Variable selbst war der Funktion nicht bekannt. Stattdessen sollte man, um ein unnötiges Kopieren der Datenstruktur zu vermeiden, auch hier mit „call-by-reference“ arbeiten, dabei aber über das Schlüsselwort `const` sicherstellen, dass keine Schreibzugriffe auf die Variable möglich sind.

Der Funktion zur Ausgabe des Spielfelds hätte zum Beispiel folgende Signatur (Signatur nennt man i. A. den Namen einer Funktion, zusammen mit den Eingangs- und Ausgangsdatentypen):

```
void SpielfeldAnzeigen(const Spielfeld & feld);
```

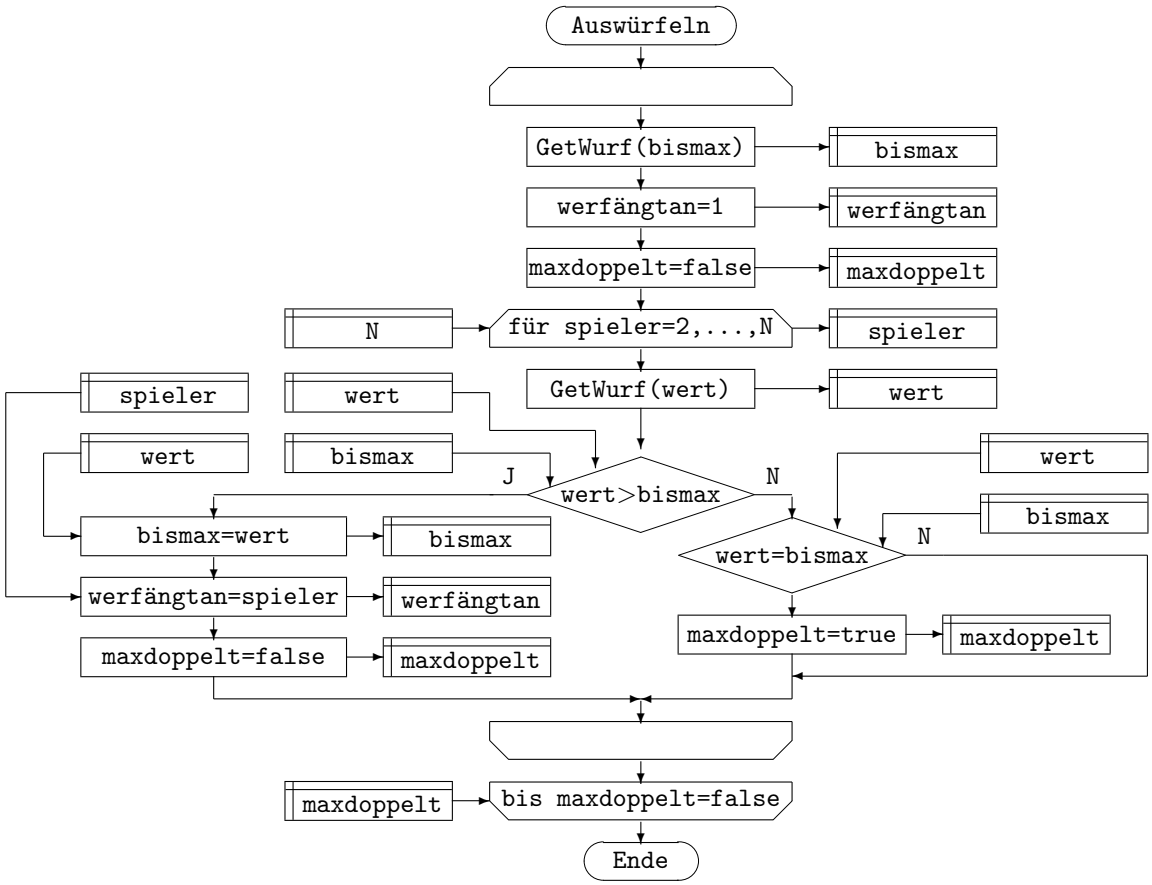
Zur Ausgabe des Spielfelds reicht es als Parameter die „linke Hälfte“ der gesamten Datenstruktur zu übergeben (Datentyp `Spielfeld`). Da bei der Ausgabe ausschließlich lesend auf die Daten zugegriffen werden soll, markieren wir den Parameter als `const`. Statt den Compiler anzuweisen, die Daten vorher zu kopieren, um dann während der Ausgabe auf der Kopie zu arbeiten, reicht es, der Ausgabefunktion mitzuteilen, an welcher Speicheradresse das Spielfeld abgelegt ist – wir benutzen einen Referenzparameter – haben den ausschließlichen Lesezugriff

trotzdem durch das `const` garantiert. Wann immer Sie bei Funktionsaufrufen Variablen übergeben wollen, die einen zusammengesetzten Typ haben (wie z. B. Verbunde), sollten Sie aus Effizienzgründen die Kombination aus `const` und Referenzparameter (`&`) wählen. Wenn der Parameter selbst Ergebnis einer Funktion ist, ist dies jedoch nicht möglich (selbst überlegen, was wäre, wenn man den Term `x+y` als Referenzparameter zu übergeben versucht).

1.6.7 Programmablaufplan mit Daten (PAD), Programmnetz (PN) und Datenflussdiagramme (DF) höherer Ebenen

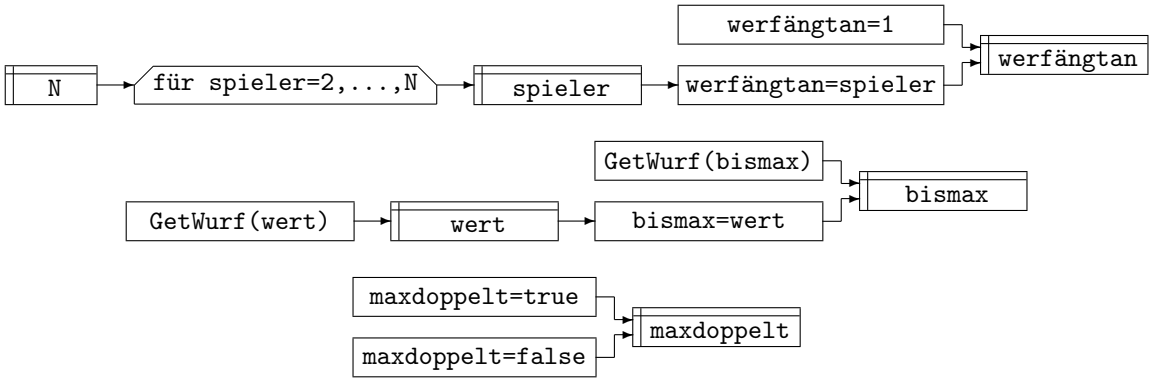
Prinzipiell haben wir nun Diagramme für jede der vier Sichtweisen, Ablauf/Fluss und Hierarchie der Programme und zugehöriger Daten, kennengelernt. Die am Anfang von Abschnitt 1.6 genannten PAD und PN fehlen jedoch noch. Diese stellen eine Zusammenfassung von PA und DF in einem Diagramm dar. Da diese oft – selbst bei kleineren Problemstellungen – zur Unübersichtlichkeit neigen, stellen wir sie hier nur kurz vor.

Als Beispiel betrachten wir hier nochmals das Teilproblem „Auswürfeln“ aus dem Mensch-Ärgere-Dich-Nicht-Spiel. Um aus einem PA einen PAD zu erhalten, fügen wir an jedem Befehl die Ein- und Ausgangsdaten für diesen Befehl als Daten-Kästchen (wie aus den bisherigen DF bekannt) hinzu. Dabei gilt die Grundregel für die Darstellung: bei lesenden Zugriffen Daten-Kästchen links von der Anweisung, bei schreibenden rechts davon (in der Abbildung unten sind Abweichungen davon nur aus Platzgründen erfolgt). Wir erhalten folgenden Programmablaufplan mit Daten (PAD) für das Teilproblem „Auswürfeln“.



Beim Programmnetz (PN) wird für jede Variable im gesamten Diagramm nur ein Daten-Rechteck eingefügt und die Verbindungen für den Datenfluss zu jedem Befehl entsprechend gezogen (ein PN ist also ein PAD, bei dem nicht jeder Befehl eigene Daten-Rechtecke bekommt). Auch bei diesem kleinen Beispiel wird das Diagramm aufgrund der vielen sich dann überschneidenden Linien unübersichtlich – wir verzichten hier auf eine separate Abbildung. Programmnetze sind meist nur auf sehr abstrakter Ebene sinnvoll, wenn die Grobstruktur eines Projekts skizziert werden soll und dabei die meisten Daten jeweils nur in wenigen (und zudem nah beeinanderliegenden) Teilen benutzt werden.

Darüberhinaus bildet das PN die Grundlage für Datenflussdiagramme höherer Ebenen: Datenflussdiagramme haben wir bisher nur zur Darstellung der Schnittstelle benutzt (eine abstrakte Anweisung mit den zugehörigen Ein- und Ausgangsdaten). Im Allgemeinen kann und will man mit Datenflussdiagrammen auch die Zwischenschritte der Veränderung von Daten innerhalb einer gesamten Abstraktionsebene visualisieren. Man nimmt dazu das zugehörige PN und streicht den kompletten Kontrollfluss (also die Pfeile, die dem PA entstammen) sowie Fallunterscheidungen und die Bedingungen aus Schleifen heraus. In unserem Beispiel verbleiben so lediglich 6 Daten-Rechtecke für die 6 Variablen, die Befehlskästchen, die auf Variablen schreibend zugreifen[3] und die Verbindungen zwischen Daten- und Befehlskästchen (alle anderen Verbindungen entfallen).



Die doppelt vorhandene Anweisung `maxdoppelt=false` wird nur ein Mal ins DF aufgenommen (man sieht den Befehlen im DF sowieso nicht mehr an, wann sie ausgeführt werden). An dem so erstellten DF erkennt man z. B., dass die entsprechenden Variablen parallel verarbeitet werden und sich nur mittelbar gegenseitig beeinflussen. Andererseits stellt das DF dann nur noch mögliche Datenflüsse dar – die Bedingungen, unter denen dieses geschieht, sind im DF nicht mehr sichtbar: man sieht zwar, dass die Variable `spieler` nur im Rahmen der Zuweisung an die Variable `werfängtan` weiterverarbeitet wird und dass der Variablen `maxdoppelt` nur die Konstanten `true` und `false` zugewiesen werden, aber nicht, ob dies immer der Fall ist, ggf. in welcher Reihenfolge oder nur in bestimmten Situationen.

1.6.8 Abstrakte Datentypen (ADT) und Modularisierung

Unser Projekt Mensch-Ärgere-Dich-Nicht nimmt langsam einen größeren Umfang an - rekapitulieren wir zunächst nochmal die bereits vorhandenen (und schon angesprochenen) Komponenten.

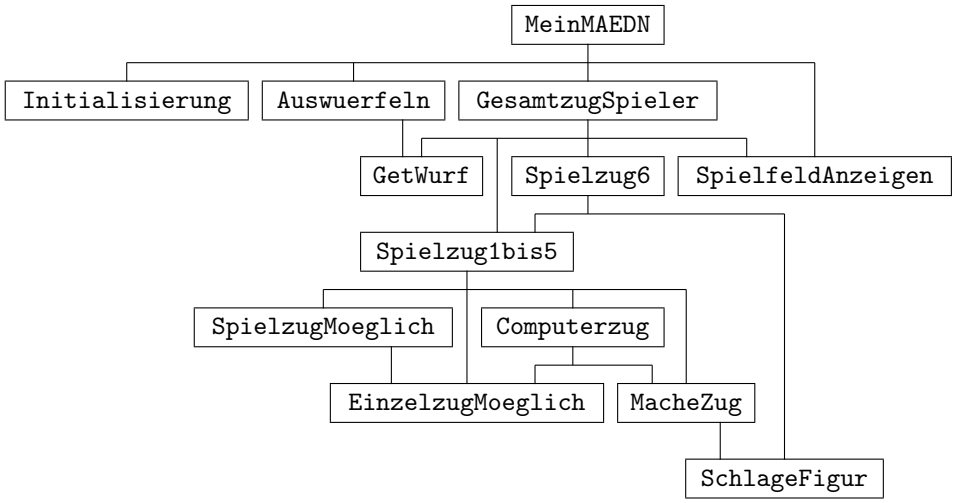
[3] Prinzipiell können auch Befehle, die nur Lesezugriff haben, im DF auftauchen; ein typisches Beispiel wäre die Ausgabe von Daten.

- GetWurf – Erzeugen einer Zufallszahl zwischen 1 und 6
- Auswürfeln – Wer beginnt?
- Gesamtzug eines Spielers – Wiederholte Einzelzüge abhängig vom Würfeln einer 6 und der Anzahl der Figuren auf dem Feld; verwendet die noch nicht näher beschriebenen Funktionen für die Teilaufgaben
  - Einzelzug, wenn eine 6 gewürfelt wurde
  - Einzelzug, wenn keine 6 gewürfelt wurde (oder wenn keine Sonderbehandlung für eine gewürfelte 6 nötig ist)
- Datenstruktur `MAEDN` – Verbund für Spielfeld und Spieler samt deren weiterer Attribute

Folgende Komponenten fehlen noch:

- Das „Hauptprogramm“, das die Einzelfunktionen koordiniert
- Initialisierung der Datenstruktur zu Beginn des Spiels
- Anzeige des Spielfelds
- Weitere Funktionen für Teilaufgaben innerhalb der Einzelzüge:
  - Test, ob mit einem gegebenen Wurf eine bestimmte Figur oder überhaupt eine Figur gezogen werden kann
  - Ziehen einer bestimmten Figur mit einem gegebenen Wurf
  - Schlagen einer (gegnerischen) Figur – beinhaltet insbesondere die Anpassung der Datenstruktur bzgl. der geschlagenen Figur
- Ein einfacher Computerspieler

Das zugehörige Programmhierarchiediagramm hat nun auch einige Ebenen bekommen.



Wenn mehrere Personen an solch einem Projekt arbeiten sollen, kann das Projekt nicht mehr nur aus einer Datei bestehen. Ziel ist, die durch die Datenflussdiagramme und Programmhierarchiediagramme geschaffenen Schnittstellen zu benutzen, so dass die dadurch zu definierenden einzelnen Module unabhängig voneinander implementiert werden können.

Meist wird das Datenhierarchiediagramm und damit die Datenstrukturen als erstes umgesetzt, da die meisten Module diese verwenden werden müssen (in unserem Fall kommt lediglich das Auswürfeln ohne die Informationen über das konkrete Aussehen des Spielfeldes aus).

Ein Standardvorgehen zur Modularisierung fasst jeweils eine Datenstruktur und die darauf arbeitenden Funktionen zu einem Modul zusammen. Dieses – eine Datenstruktur und die darauf arbeitenden Funktionen – nennt man auch einen abstrakten Datentyp (ADT) (die genaue Bedeutung des Begriffs ADT untersuchen wir in Abschnitt 1.8). Die Funktionen, die nicht auf problemspezifischen Datenstrukturen arbeiten, kann man in ein eigenes Modul (oder thematisch gruppiert auch in mehrere) zusammenfassen. In unserem Projekt könnten so folgende drei Module entstehen:

- Ein Modul „Spielfeld“ – dies enthält die Datenstruktur für das Spielfeld und die Funktion zu dessen Anzeige
- Ein Modul „Hilfsfunktionen“ – dies enthält die von den Datenstrukturen unabhängigen Funktionen zum Würfeln und Auswürfeln, welcher Spieler beginnt
- Ein Modul „Mensch-Ärgere-Dich-Nicht“ – dies enthält die eigentlichen Funktionen zum Durchführen der Züge

Ziel bei solchen Aufteilungen in Module sind stets:

- Bei Arbeit im Team eine Aufteilung in unabhängig voneinander zu realisierende Aufgaben
- Mögliche einfache Wiederverwendung von Code in anderen Projekten (typische Beispiele hierfür sind Such- und Sortierfunktionen mit den zugehörigen Datenstrukturen (wir werden uns im zweiten Semester ausführlich mit diesen Themen beschäftigen) und Standarddatenstrukturen (wie z. B. Listen, die wir in Abschnitt 1.8 behandeln werden))

Wie werden die einzelnen Module implementiert und wie werden sie zu einem Gesamtprogramm zusammengefügt? Wir haben das Einbinden von Modulen schon vielfach verwendet – wenn auch nicht unbedingt bewusst wahrgenommen. Jedes Modul setzt sich zusammen aus einer Headerdatei (Dateiendung `.h`) und der Implementierung in C/C++ (Dateiendung `.cpp`). Das Einbinden eines Moduls geschieht über die Präprozessoranweisung `#include` – dieses haben wir z. B. beim Einbinden anderer Module zur Textausgabe schon kennengelernt. Das Übersetzen der einzelnen Module eines Projektes übernimmt der Compiler – auf die Details wird im Labor noch näher eingegangen.

Auf einen Aspekt wollen wir hier noch konkret eingehen: In C/C++ muss man darauf achten, dass eine Variable oder Funktion nicht mehrfach deklariert werden darf. Das heißt insbesondere, dass wir eigentlich ein Modul nicht mehrfach einbinden dürften – dies lässt sich i. A. aber kaum verhindern (viele Module werden z. B. das Modul zur Textausgabe benutzen wollen). Wir brauchen daher einen Mechanismus, der beim ersten Einbinden eines Moduls die Deklarationen durchführt, gleichzeitig aber bei einem weiteren Einbinden desselben Moduls verhindert, dass die Deklarationen erneut bearbeitet werden. Dazu bedient man sich auch hier Präprozessor-Anweisungen: Jede Header-Datei sollte folgenden Aufbau haben:

```
#ifndef NAMEDESMODULS_H
#define NAMEDESMODULS_H

... // Deklarationen der Datenstruktur
... // Deklarationen der bekanntzumachenden Funktionen (das sind
    // die Funktionen aus diesem Modul, die man im dem Programm,
    // das dieses Modul einbindet, verwenden können soll)

#endif
```

`NAMEDESMODULS` wird dabei natürlich durch den konkreten Dateinamen ersetzt (die Großschreibung ist üblich, aber nicht zwingend notwendig).

Die Funktionsweise: Der Befehl `#ifndef` fragt ab, ob dem Compiler schon ein Bezeichner `NAMEDESMODULS_H` bekannt ist (`ifndef`=„if not defined“), wobei als Wahrheitswert `true` zurückgegeben wird, wenn der Bezeichner noch nicht bekannt ist, und `false` sonst. Wird `true` zurückgeliefert, so werden die folgenden Zeilen bearbeitet und dabei über `#define` der besagte Bezeichner bekannt gemacht; wird `false` zurückgeliefert, so werden die Zeilen bis zum `#endif` übersprungen.

Dies hat zur Folge, dass beim ersten Einbinden das `#ifndef` den Wert `true` liefert und so die Deklarationen bearbeitet werden; bei jedem weiteren Einbinden ist der Bezeichner `NAMEDESMODULS_H` bekannt und die Deklarationen werden übersprungen.

1.6.9 Der Programmcode „Mensch-Ärgere-Dich-Nicht“ und Known Bugs

In diesem Abschnitt befinden sich der Source-Code der auf sieben Dateien verteilten Module:

- `spielfeld.h` / `spielfeld.cpp` – Datenstruktur für das Spielfeld und eine rudimentäre Routine zum Anzeigen
- `maednhelpers.h` / `maednhelpers.cpp` – Hilfsroutinen zum Würfeln und Auswürfeln, wer beginnt
- `maedn.h` / `maedn.cpp` – die wesentlichen Routinen des Spiels, insbesondere alles zum Überprüfen und Durchführen eines Zuges sowie ein simpler Computerspieler
- `menschaergeredichnicht.cpp` – das „`int main()`“ – dies bindet hier letzten Endes nur die benötigten Module ein und ruft das Spiel auf

In den Kommentarzeilen finden Sie noch viele weitere Hinweise, wo welche Teile noch verändert werden sollten und wo noch Teile fehlen (z. B. noch nicht berücksichtigte Regeln oder nicht beachtete Sonderfälle). Sie finden die Dateien auch auf der Webseite zur Vorlesung. Versuchen Sie die fehlenden Teile zu ergänzen.

Die Datei `spielfeld.h`

```
#ifndef SPIELFELD_H
#define SPIELFELD_H

#include <iostream>  // für die einfache Textausgabe

const int MAX_Spieler=4;
const int MAX_Felder=10*MAX_Spieler;
typedef bool Zielfelder[4]; // Typ für den Zielbereich
struct Spielfeld {
  int start[MAX_Spieler];       // Anzahl Figuren, die nicht draußen sind
  int besetzt[MAX_Felder];      // Rundstrecke ab 1. Spieler gezählt, Wert=spielernr
  Zielfelder ziel[MAX_Spieler]; // Zielbereiche der einzelnen Spieler
} ;

void SpielfeldAnzeigen(const Spielfeld &);

#endif
```

Die Datei `spielfeld.cpp`

```
#include "spielfeld.h"

using namespace std; // für iostream

////////////////////////////////////////////////////////////////////////////
void SpielfeldAnzeigen(const Spielfeld & feld) {                          //
////////////////////////////////////////////////////////////////////////////
  // Das kann man bestimmt auch schöner machen ...
  // ist so aber für beliebige Werte MAX_Spieler funktionsfähig
  int i,j;
  cout << "Aktuelle Belegung des Spielfelds:" << endl;
  // erst die Startbereiche
  for(i=0;i<MAX_Spieler;i++) {
    for(j=0;j<feld.start[i];j++) {
      cout << i+1; // +1, Spielernummern so 1,...,4
    }
    for(;j<11;j++) {
      cout << ' ';
    }
  }
  cout << endl;
  // nun das Spielfeld in 10er Abschnitten
  for(i=0;i<MAX_Spieler;i++) {
    cout << ' ';
    for(j=0;j<10;j++) {
      cout << 1+feld.besetzt[i*10+j];
```

```
    }
  }
  cout << endl;
  // und die Zielbereiche
  for(i=0;i<MAX_Spieler;i++) {
    for(j=0;j<4;j++) {
      if (feld.ziel[i][j]) {
        cout << i+1; // +1, Spielernummern so 1,...,4
      } else {
        cout << 0;
      }
    }
    cout << "       "; // 7 Blanks
  }
  cout << endl;
}
```

Die Datei `maednhelpers.h`

```
#ifndef MAEDNHELPERS_H
#define MAEDNHELPERS_H

#include<cstdlib>

void GetWurf(int &);
void Auswuerfeln(const int, int &);

#endif
```

Die Datei `maednhelpers.cpp`

```
#include "maednhelpers.h"

////////////////////////////////////////////////////////////////////////////
void GetWurf(int & wurf) { // Zufallszahl zwischen 1 und 6                  //
////////////////////////////////////////////////////////////////////////////
  wurf = rand() % 6 +1; // rand() liefert eine ganzzahlige Zufallszahl,
  // das % rechnet diese modulo 6 (Werte zwischen 0 und 5, deshalb noch +1)
}

////////////////////////////////////////////////////////////////////////////
void Auswuerfeln(const int N, int & werfaengtan) {                         //
////////////////////////////////////////////////////////////////////////////
  // gegenüber alter Version etwas angepasst: kleinste Nr ist 0
  // muss ggf. noch angepasst werden, wenn spieltmit=0 vorkommt
  int bismax, spieler, wert;
  bool maxdoppelt;
  do {
```

```
    GetWurf(bismax);
    werfaengtan=0;
    maxdoppelt=false;
    for(spieler=1;spieler<N;spieler++) {
      GetWurf(wert);
      if (wert>bismax) {
        bismax=wert;
        werfaengtan=spieler;
        maxdoppelt=false;
      } else {
        if (wert==bismax) {
          maxdoppelt=true;
        }
      }
    }
  } while (!(maxdoppelt==false));
}
```

Die Datei `maedn.h`

```
#ifndef MAEDN_H
#define MAEDN_H

#include "spielfeld.h"
#include "maednhelpers.h"
#include <iostream>  // für die einfache Textausgabe

using namespace std; // für iostream

struct Spieler {
  int spieltmit;     // 0=leer, 1=Mensch, 2=Computer
  char name[20];     // Name
  int aufFeld;       // Anzahl Figuren auf dem Feld
  int imZiel;        // Anzahl Figuren im Zielbereich
  int pos[4];        // Positionen seiner Figuren bzgl. seinem Startfeld
};
struct MAEDN {
  Spielfeld feld;
  Spieler spieler[MAX_Spieler];
};

// man braucht hier nach außen nur MeinMAEDN(void) sichtbar zu machen
// die anderen Funktionen sollen ja garnicht einzeln verwendet werden
// falls doch, einfach entsprechende Funktionen entkommentieren
//void Initialisierung(MAEDN &);
//bool EinzelzugMoeglich(const MAEDN &, const int, const int, const int);
//bool SpielzugMoeglich(const MAEDN &, const int, const int);
//void SchlageFigur(MAEDN &, const int);
```

```
//void MacheZug(MAEDN &, const int, const int, const int);
//void Computerzug(MAEDN &, const int, const int);
//void Spielzug1bis5(MAEDN &, const int, const int);
//void Spielzug6(MAEDN &, const int);
//void GesamtzugSpieler(MAEDN &, const int);
void MeinMAEDN(void);

#endif
```

Die Datei `maedn.cpp`

```
#include "maedn.h"

using namespace std; // für iostream

////////////////////////////////////////////////////////////////////////////
void Initialisierung(MAEDN & m) {                                         //
// Init. der Teil-Datenstruktur Spielfeld, Einlesen der Spielernamen etc.  //
// statt Einlesen, bisher nur Default-Werte                                //
////////////////////////////////////////////////////////////////////////////
  // Werte richtig initialisieren
  int i,j;
  for(i=0;i<MAX_Spieler;i++) { // 4 Figuren je Startbereich
    m.feld.start[i]=4;
  }
  for(i=0;i<MAX_Felder;i++) { // alle Felder unbesetzt
    m.feld.besetzt[i]=-1;
  }
  for(i=0;i<MAX_Spieler;i++) { // alle Zielbereiche leer
    for(j=0;j<4;j++) {
      m.feld.ziel[i][j]=false;
    }
  }
  // per Default alles Computerspieler bis auf den ersten
  // ggf. ausprogrammieren und abfragen
  // oder "Farben" zufällig zuordnen
  // möglich auch, eine "Farbe" leer zu lassen (spieltmit=0)
  for(i=0;i<MAX_Spieler;i++) {
    m.spieler[i].spieltmit=2; // default Computerspieler
    strcpy(m.spieler[i].name,"Computer\0");
    m.spieler[i].aufFeld=0;
    m.spieler[i].imZiel=0;
    for(j=0;j<4;j++) {
      m.spieler[i].pos[j]=-1; // noch nicht draußen
    }
  }
  m.spieler[0].spieltmit=1; // der erste Spieler ist Mensch
  strcpy(m.spieler[0].name,"Spieler\0"); // und heißt "Spieler"
```

```
  //
  cout << "In der Initialisierungsroutine wurden Spieler 2 und 4 auf "
       << "'spielt nicht mit' gesetzt ..." << endl;
  m.spieler[1].spieltmit=0;
  m.spieler[3].spieltmit=0;
}

////////////////////////////////////////////////////////////////////////////
bool EinzelzugMoeglich(const MAEDN & m, const int nr, const int figur, const int wert) {
// überprüft, ob Spieler <nr> die <figur> mit Wurf <wert> ziehen kann        //
////////////////////////////////////////////////////////////////////////////
  // ausnahmsweise Abweichung von der Standardumsetzung!
  // das Ergebnis des Tests wird mit return zurückgegeben
  // sonst müsste man für das Ergebnis noch einen formalen Parameter haben
  // ebenso in der aufrufenden Funktion, die Abfrage in der
  // if-Anweisung sähe dann so aus:
  // if (EinzelzugMoeglich(m,nr,figur,wert,erg),erg)
  // also zunächst der Aufruf und durch Verwendung des Komma-Operators
  // ist der Wahrheitswert der Bedingung der Inhalt der Variablen erg
  // Dieses Vorgehen bietet sich an, wenn das Ergebnis ausschließlich direkt
  // verwendet wird und nie einer Variablen zugewiesen werden soll.
  // Man kann dies auch als "anonyme" Ausgangsvariable (ohne Namen) betrachten
  //
  // Einzelzug mit der bestimmten Figur möglich, wenn
  // - Figur draußen
  // - das Feld "wert" weiter existiert
  // - und nicht durch eine eigene Figur besetzt ist
  // Es fehlt noch (falls gewünscht) Überprüfung, dass im Zielbereich keine
  // eigenen Figuren übersprungen werden
  if (m.spieler[nr].pos[figur]>=0) {
    if (m.spieler[nr].pos[figur]+wert<MAX_Felder+4) {
      if (m.spieler[nr].pos[figur]+wert<MAX_Felder) {
        if (m.feld.besetzt[(10*nr+m.spieler[nr].pos[figur]+wert)%MAX_Felder]!=nr) {
          return true; // im Feld und frei oder gegnerische Figur
        } else {
          return false; // im Feld, aber eigene Figur
        }
      } else {
        return !(m.feld.ziel[nr][m.spieler[nr].pos[figur]+wert-MAX_Felder]);
        // es ist bereits ein bool-Wert,
        // Rückgabe true -> noch frei, false -> schon besetzt
      }
    } else {
      return false; // Zug schießt über das Ziel hinaus
    }
  } else {
    return false; // Figur ist noch im Startbereich
  }
}
```

```
////////////////////////////////////////////////////////////////////////////
bool SpielzugMoeglich(const MAEDN & m, const int nr, const int wert) {       //
// überprüft, ob Spieler <nr> mit Wurf <wert> eine Figur setzen kann         //
////////////////////////////////////////////////////////////////////////////
  if (EinzelzugMoeglich(m,nr,0,wert)) {
    return true;
  } else if (EinzelzugMoeglich(m,nr,1,wert)) {
    return true;
  } else if (EinzelzugMoeglich(m,nr,2,wert)) {
    return true;
  } else if (EinzelzugMoeglich(m,nr,3,wert)) {
    return true;
  } else {
    return false;
  } // dies ist die ausführliche Variante von
  // return ( EinzelzugMoeglich(m,nr,0,wert) || EinzelzugMoeglich(m,nr,1,wert)
  //       || EinzelzugMoeglich(m,nr,2,wert) || EinzelzugMoeglich(m,nr,3,wert) );
}

////////////////////////////////////////////////////////////////////////////
void SchlageFigur(MAEDN & m, const int index) {                              //
// Schlägt die Figur auf dem Feld <index>, das Feld darf nicht leer sein     //
////////////////////////////////////////////////////////////////////////////
  int gegner,i;
  gegner=m.feld.besetzt[index];
  cout << "Die Figur von " << m.spieler[gegner].name
       << " fliegt raus!" << endl;
  m.feld.start[gegner]++;
  m.spieler[gegner].aufFeld--;
  for(i=0;i<4;i++){ // vielleicht nicht so schön, dass man suchen muss,
    // welche der gegenerischen Figuren rausflog.
    // Evtl. kann man das noch in die Figuren codieren,
    // ist aber etwas aufwendig und an einigen Stellen zu ändern
    // => wir lernen an diesem Beispiel: Konstruktion einer
    // geeigneten Datenstruktur ist schwierig. Man hat selten zu
    // Beginn gleich alles mögliche im Sinn.
    // Spätere Änderungen sind immer aufwendig,
    // eine detaillierte Planung zahlt sich aus
    if (((m.spieler[gegner].pos[i]+10*gegner)%MAX_Felder) == index) {
      m.spieler[gegner].pos[i]=-1;
      break; // es kann nur eine geben ...
    }
  }
}

////////////////////////////////////////////////////////////////////////////
void MacheZug(MAEDN & m, const int nr, const int figur, const int wert) {   //
// Führt den Zug für Spieler <nr> mit <figur> und Wurf <wert> aus            //
```

```
// darf nur aufgerufen werden, wenn der Spielzug möglich ist!                //
////////////////////////////////////////////////////////////////////////////
  // Es sind mehrere Fälle zu unterscheiden:
  // Lag die alte Position auf der Rundstrecke oder im Ziel
  // Liegt die neue Position auf der Rundstrecke oder im Ziel
  if (m.spieler[nr].pos[figur]+wert<MAX_Felder) {
    // (alte und) neue Position ist noch auf der Rundstrecke,
    // evtl. wird gegnerische Figur geschlagen
    if (m.feld.besetzt[(10*nr+m.spieler[nr].pos[figur]+wert)%MAX_Felder]!=-1) {
      SchlageFigur(m,(10*nr+m.spieler[nr].pos[figur]+wert)%MAX_Felder);
    } // else { ... }
    // Der eigene Zug ist unabhängig davon auszuführen
    m.feld.besetzt[(10*nr+m.spieler[nr].pos[figur])%MAX_Felder]=-1;
    m.feld.besetzt[(10*nr+m.spieler[nr].pos[figur]+wert)%MAX_Felder]=nr;
    m.spieler[nr].pos[figur]+=wert; // den Wert kann man einfach erhöhen :)
  } else {
    // Figur ist danach im Zielbereich,
    // gegnerische Figur kann nicht geschlagen werden
    if (m.spieler[nr].pos[figur]<MAX_Felder) {
      // Figur war vorher draußen, jetzt aber im Zielbereich
      m.feld.besetzt[(10*nr+m.spieler[nr].pos[figur])%MAX_Felder]=-1;
      m.spieler[nr].pos[figur]+=wert; // den Wert kann man einfach erhöhen :)
      m.feld.ziel[nr][m.spieler[nr].pos[figur]-MAX_Felder]=true;
      m.spieler[nr].aufFeld--;
      m.spieler[nr].imZiel++;
    } else {
      // Figur war vorher und ist nachher im Zielbereich
      m.feld.ziel[nr][m.spieler[nr].pos[figur]-MAX_Felder]=false;
      m.spieler[nr].pos[figur]+=wert; // den Wert kann man einfach erhöhen :)
      m.feld.ziel[nr][m.spieler[nr].pos[figur]-MAX_Felder]=true;
    }
  }
}

////////////////////////////////////////////////////////////////////////////
void Computerzug(MAEDN & m, const int nr, const int wert) {                //
// darf nur aufgerufen werden, wenn <nr> überhaupt einen Zug machen kann   //
// Einfacher automatischer Spieler:                                        //
//   - wenn Figur auf Startposition, setze diese weg, wenn möglich         //
//   - sonst wähle erste Figur (nach interner Zählung), die möglich ist    //
////////////////////////////////////////////////////////////////////////////
  int figur;
  figur=0;
  if (m.feld.besetzt[10*nr]==nr && m.feld.besetzt[10*nr+wert]!=nr) {
    // welche der 4 Figuren steht denn auf der Startposition?
    while (m.spieler[nr].pos[figur]!=0) {
      figur++;
    }
  } else { // suche erste mögliche Figur
```

```
    while (!(EinzelzugMoeglich(m,nr,figur,wert))) {
      figur++;
    }
  }
  // und nun noch den Zug ausführen ...
  cout << m.spieler[nr].name << " sagt: Ich setze Figur " << figur << endl;
  MacheZug(m,nr,figur,wert);
}

////////////////////////////////////////////////////////////////////////////
void Spielzug1bis5(MAEDN & m, const int nr, const int wert) {               //
// Führt den Zug für Spieler <nr> aus, der eine <wert> gewürfelt hat        //
// (auch, wenn <nr> eine 6 gewürfelt hat und keine Figur raussetzen konnte) //
////////////////////////////////////////////////////////////////////////////
  int i, figur;
  if (SpielzugMoeglich(m,nr,wert)) {
    cout << "Du hast eine " << wert << " gewuerfelt." << endl;
    if (m.spieler[nr].spieltmit!=1) { // dann ist es ein Computerspieler
      Computerzug(m,nr,wert);
    } else {
      // Hier müss(t)en nun alle Sonderregeln abgefangen werden, wie z.B.
      // Figur auf Startfeld muss weggezogen werden,
      //   wenn noch weitere Figuren im Startbereich sind und der Zug möglich ist
      // ggf. Rauswerfpflicht?
      // kein Überspringen der eigenen Figuren im Zielbereich
      //   (wobei das eher in EinzelzugMoeglich gehört)
      do {
        cout << "Welche Figur willst Du ziehen?" << endl;
        for(i=0;i<4;i++) {
          if (EinzelzugMoeglich(m,nr,i,wert)) {
            cout << "Figur " << i << " steht von Deiner Startposition "
                 << "aus auf Feld " << m.spieler[nr].pos[i] << endl;
          }
        }
        cout << "Ich nehme Figur (eine der moeglichen Nummern auswaehlen) ";
        cin >> figur;
        // statt der Variablen figur könnte man auch i nehmen,
        // aber Lesbarkeit geht vor (vermeintlicher) Speichereffizienz
        // Aufgabe: was passiert, wenn der Spieler einen Wert <0 || >3 angibt?
        // Fangen Sie solche Eingaben noch ab!
      } while (!(EinzelzugMoeglich(m,nr,figur,wert))); // Wiederholung,
      // bis er endlich eine der möglichen Figuren gewählt hat
      //
      // und nun müssen wir den Zug noch ausführen ...
      MacheZug(m,nr,figur,wert);
    }
  } else { // gehört zu if (SpielzugMoeglich(m,nr,wert))
    cout << "Schade, kein Zug mit einer " << wert << " moeglich." << endl;
  }
```

```
}

////////////////////////////////////////////////////////////////////////////
void Spielzug6(MAEDN & m, const int nr) {                                 //
// Führt den Zug für Spieler <nr> aus, der eine 6 gewürfelt hat           //
////////////////////////////////////////////////////////////////////////////
  int i;
  // Zwangszug bei 6: Figur raussetzen, falls möglich
  if (m.feld.start[nr]>0 && m.feld.besetzt[10*nr]!=nr) {
    // noch Figuren am Start und Startfeld nicht durch eigene Figur besetzt
    cout << "Du hast eine 6 gewuerfelt! "
         << "Glueckwunsch! Neue Figur ist draussen!" << endl;
    if (m.feld.besetzt[10*nr]!=-1) {
      SchlageFigur(m,10*nr);
    } // else { ... }
    // Der eigene Zug ist unabhängig davon auszuführen
    m.feld.start[nr]--;
    m.spieler[nr].aufFeld++;
    // erste Figur finden, die noch draußen ist, auch das geht bestimmt schöner
    i=0;
    while (m.spieler[nr].pos[i]!=-1) { // mind. eine IST noch draußen
      i++;
    }
    m.feld.besetzt[10*nr]=nr;
    m.spieler[nr].pos[i]=0;
  } else { // alle anderen Sonderfälle wie bei 1 bis 5
    Spielzug1bis5(m,nr,6); // etwas gemogelt :) - ist aber korrekt und kurz
    // sonstige Sonderfälle werden in Spielzug1bis5 behandelt
  }
}

////////////////////////////////////////////////////////////////////////////
void GesamtzugSpieler(MAEDN & m, const int nr) {                          //
// Gesamtzug für Spieler <nr> inkl. Wiederholungen nach 6 bzw. 3* am Anfang //
////////////////////////////////////////////////////////////////////////////
  int count, wert;
  cout << "Spieler " << m.spieler[nr].name
       << " (Figuren-Nummer " << nr+1 << ") ist dran" << endl;
  if (m.spieler[nr].aufFeld==0) { // die Abfrage ist nicht korrekt
    // wenn keine Figur auf dem Feld ist, kann trotzdem noch ein Zug mit
    // einer Figur innerhalb des Zielbereichs möglich sein
    count=0;
    do {
      count++;
      GetWurf(wert);
      cout << count << "-ter Wurf: eine " << wert << " ..." << endl;
    } while (!(count==3 || wert==6));
    if (wert==6) {
      do {
```

```
        Spielzug6(m,nr);
        SpielfeldAnzeigen(m.feld);
        GetWurf(wert);
      } while (!(wert!=6)); // ok, hier würde man zu wert==6 vereinfachen :)
      Spielzug1bis5(m,nr,wert);
      SpielfeldAnzeigen(m.feld);
    }
  } else {
    GetWurf(wert);
    while (wert==6) {
      Spielzug6(m,nr);
      SpielfeldAnzeigen(m.feld);
      GetWurf(wert);
    }
    Spielzug1bis5(m,nr,wert);
    SpielfeldAnzeigen(m.feld);
  }
}

////////////////////////////////////////////////////////////////////////////
void MeinMAEDN(void) { // das "Hauptprogramm" für das MÄDN-Spiel             //
////////////////////////////////////////////////////////////////////////////
  MAEDN m;
  int nr;
  Initialisierung(m);
  do {
    Auswuerfeln(MAX_Spieler,nr);
  } while (!(m.spieler[nr].spieltmit!=0)); // kleine Mogelpackung,
  // das Auswürfeln wird wiederholt, falls ein Spieler ermittelt wird,
  // der garnicht mitmacht. Die do-while-Schleife wieder entfernen,
  // nachdem die Auswuerfeln-Routine angepasst wurde!
  cout << "Der Spieler " << m.spieler[nr].name << " faengt an!" << endl; //
  SpielfeldAnzeigen(m.feld);
  nr--; // kleiner Trick
  do { // damit hier einheitlich der nächste Spieler bestimmt werden kann
    nr=(nr+1)%MAX_Spieler;
    if (m.spieler[nr].spieltmit>0) { // Mensch oder Computer
      GesamtzugSpieler(m,nr);
    }
  } while (!(m.spieler[nr].imZiel==4));
  cout << "Glueckwunsch! " << m.spieler[nr].name << " hat gewonnen!" << endl;
}
```

Die Datei `menschaergeredichnicht.cpp`

```
#include <iostream>  // für die einfache Textausgabe
#include <ctime>     // für time() zur Initialis. des Zufallszahlengenerators
#include <cstdlib>   // für srand() und rand() für Zufallszahlen
```

```
#include "maedn.h"

using namespace std; // für iostream

int main(void) {
  // einmal(!) im Hauptprogramm für die Initialisierung
  srand(unsigned(time(0))); //  des Zufallszahlengenerators aufrufen
  // Aufruf
  MeinMAEDN();
  // zum Abschluss noch das fehlerfreie Beenden an die Konsole melden
  return 0;
}
```

Known Bugs Bei jedem Programm wird man – egal, wie sorgfältig zuvor geplant wurde – nach einiger Zeit Fehler finden. Bei obigem Programm sind noch mindestens zwei Fehler versteckt:

- Zum Teil erlaubt das Programm ein dreimaliges Würfeln, obwohl prinzipiell noch Züge mit den Figuren außerhalb des Startbereichs möglich sind. Finden Sie durch Probieren des Spiels heraus, wann dieses Phänomen auftritt, und überlegen Sie sich, an welchen Stellen sinnvoll die notwendigen Änderungen eingefügt werden sollten.
- Recht selten tritt das Phänomen auf, dass eine fünfte Figur eines Spielers angezeigt wird, die bei der Angabe der möglichen Züge nicht berücksichtigt wird. Finden Sie heraus, wann dieses auftritt, worin der eigentlich Fehler besteht und wie man den Fehler beheben kann.
- Haben Sie noch weitere Fehler gefunden?

1.7 Rekursion und das Rückgabekonzept von C/C++

In der Mathematik stößt man recht häufig auf rekursive Definitionen. Als ein typisches Beispiel seien hier die Fibonacci-Zahlen $\text{fibo}(n) : \mathbb{N} \to \mathbb{N}$ genannt:

$$\text{fibo}(n) = \begin{cases} 1 & \text{, falls } n \leq 1 \\ \text{fibo}(n-1) + \text{fibo}(n-2) & \text{, falls } n > 1 \end{cases}$$

Dabei werden in der Definition der Fibonacci-Zahlen die Fibonacci-Zahlen selbst verwendet, aber mit kleinerem Argument. Die Angabe der Funktionswerte für die Argumente 0 und 1 ist hier ausreichend, um die Funktion vollständig zu definieren. Allgemein ergibt sich der Funktionswert durch Verknüpfung anderer Funktionswerte derselben Funktion oder einem Funktionswert aus einer ausreichend großen Menge fest definierter Werte.

In der Informatik wird diese Idee der Rekursion als Problemlösestrategie aufgegriffen: bei einfachen Problem-Instanzen löst man das Problem direkt (einfach heißt im Beispiel der Fibonacci-Zahlen, dass das Argument 0 oder 1 ist) oder das Problem wird auf ein oder mehrere kleinere Probleme derselben Art reduziert (im Beispiel die Summe zweier Fibonacci-Zahlen mit jeweils kleinerem Argument).

1.7.1 Einschub: das Rückgabekonzept von C/C++

Bei der Umsetzung der strukturierten Entwürfe haben wird bisher reine Eingangsparameter (Parameter mit ausschließlicher Leseberechtigung) als `const`-Parameter umgesetzt, reine Ausgangsparameter (Schreibberechtigung nötig) und Durchgangsparameter (Ausgangsparameter, die gleichzeitig Eingangsparameter sind) über das Referenzkonzept (`&` zwischen Datentyp und Bezeichner des formalen Parameters) umgesetzt. Dabei haben wir das Rückgabekonzept, das C/C++ in seinen Funktionen zur Verfügung stellt, konsequent nicht verwendet (alle Funktionen hatten das Wort `void` vorangestellt).

Der allgemeine Aufbau einer C/C++-Funktion ist

```
<Datentyp> <Funktionsbezeichner> (<Liste der formalen Parameter>) { <Kommandos> }
```

wobei der Datentyp der Typ des Rückgabewertes ist. Wir haben dieses Konzept ausnahmsweise bei zwei der Funktionen des Mensch-Ärgere-Dich-Nicht-Projektes verwendet (bei den Tests, ob ein Spielzug möglich ist). Verwendet man das Rückgabekonzept von C/C++, muss man sicherstellen, dass das Ergebnis der Funktion mit dem Befehl `return` der aufrufenden Funktion übergeben wird. Als kleines Beispiel diene hier eine Funktion `add`, die die Addition zweier Zahlen realisiert.

Bei der Umsetzung des strukturierten Entwurfs hätten wir einen dritten formalen Parameter eingeführt:

```
void add(const int a, const int b, int & c) {
  c = a + b;
}
```

Der Aufruf im Hauptprogramm wäre

```
int a,b,c;
...
add(a,b,c);
```

Mit Verwendung des Rückgabekonzepts von C/C++ hat die Funktion als Rückgabedatentyp den Typ `int`:

```
int add(const int a, const int b) {
  int c;
  c = a + b;
  return c;
}
```

Der Aufruf im Hauptprogramm wäre

```
int a,b,c;
...
c = add(a,b);
```

Der Vorteil des strukturierten Entwurfs liegt in der automatischen Umsetzbarkeit – zudem erlaubt das Rückgabekonzept von C/C++ nur einen Rückgabeparameter, wodurch man bei

mehreren Rückgabewerten wieder auf das Referenzparameterkonzept ausweicht (oder – programmiertechnisch unschön – sich extra einen `struct`-Datentyp anlegt, der die inhaltlich nicht unbedingt zusammengehörigen Datentypen zusammenfasst, um diese einer Variablen zuweisen zu können (und die Einzelwerte danach in weiteren Befehlen den eigentlichen Zielvariablen zuzuweisen)).

Zurück zum Beispiel der Fibonacci-Zahlen: Mit dem Rückgabe-Konzept von C/C++ können wir zusammen mit der Rekursion die Funktion der Fibonacci-Zahlen sehr leicht in C/C++ umsetzen (letzten Endes lediglich ein Abscheiben der mathematischen Definition):

```
unsigned int fibo(const unsigned int n) {
  if (n<=1) {
    return 1;
  } else {
    return fibo(n-1)+fibo(n-2);
  }
}
```

Übungsaufgabe: Finden Sie eine rekursive Definition der Fakultätsfunktion und setzen Sie diese in eine rekursive C/C++-Funktion um.

1.7.2 Die Türme von Hanoi

An den bisherigen Beispielen lässt sich die Mächtigkeit der Rekursion nur erahnen. Wir zeigen hier noch ein weiteres klassisches Beispiel zur Verwendung der Rekursion: Die Türme von Hanoi. Das Spiel besteht aus N verschieden großen Scheiben, die zu Beginn der Größe nach sortiert auf einem von drei Stäben liegen. Die Aufgabe ist nun, mit möglichst wenigen Zügen den Scheibenstapel auf einen der anderen beiden Stäbe zu verschieben. Dabei sind lediglich zwei Regeln zu beachten:

- es darf immer nur eine Scheibe auf einmal von einem auf einen anderen Stab verschoben werden
- es darf nie eine größere auf einer kleineren Scheibe liegen

Versucht man dies von Hand zu lösen, so findet man für $N = 3$ recht schnell die optimale Lösung: Scheibe 1 von Stab 1 auf Stab 2, Scheibe 2 von Stab 1 auf 3, Scheibe 1 von 2 auf 3, Scheibe 3 von 1 auf 2, Scheibe 1 von 3 auf 1, Scheibe 2 von 3 auf 2 und schließlich Scheibe 1 von 1 auf 2. Für größere N wird es schnell unübersichtlich.

Eine mögliche Lösungsstrategie besteht darin, eine rekursive Lösung anzugeben: wenn ein Turm aus N Scheiben von Stab A zum Stab B zu verschieben ist, verschiebe zunächst den Turm bis auf die unterste Scheibe (also einen Turm der Höhe $N-1$) auf den dritten Stab (C), verschiebe die unterste Scheibe auf den Stab B und dann den auf Stab C zwischengelagerten Turm von C auf B. Das Verschieben des Turms der Höhe $N-1$ wird durch dasselbe Verfahren durchgeführt – wenn $N = 1$ ist, brauchen wir natürlich nur die eine Scheibe direkt zu verschieben.

Als C/C++-Programm sieht dies dann so aus:

```
#include <iostream>
using namespace std;

void move(const unsigned int disc, const char start, const char end){
  cout << "Verschiebe Scheibe " << disc << " von Stab " << start
       << " auf Stab " << end << "." << endl;
}

void hanoi(const unsigned int disc, char start, char end, char third) {
if (disc == 1){
    move(1, start, end);
  } else {
    hanoi(disc-1, start, third, end); // erst Teilturm auf den 3. Stab
    move(disc, start, end);
    hanoi(disc-1, third, end, start); // dann von dort zum Zielstab
  }
}

int main(void) {
  cout << "Demo der Tuerme von Hanoi fuer N=3:" << endl;
  hanoi(3,'A','B','C'); // verschiebe den Turm der Höhe 3 von Stab A
  // nach Stab B mit Stab C als Zwischenposition
  return 0;
}
```

Übungsaufgabe: Untersuchen Sie den Ablauf des Programms mit Hilfe des Debuggers. Machen Sie sich klar, dass beim rekursiven Aufruf der Funktion stets neue lokale Variablen der Funktionsparameter erzeugt werden, so dass die alten Werte nach Rückkehr vom (rekursiven) Funktionsaufruf nicht verloren sind.

Randbemerkung: Die klassischen Beispiele zur Rekursion sind bzgl. der Laufzeit aufwendig. Während bei den Fibonacci-Zahlen leicht eine deutlich laufzeiteffizientere iterative Variante (also mit Schleifen) gefunden werden kann, ist die Lösung für die Türme von Hanoi bereits optimal (auch hier gibt es eine (wenig verbreitete) iterative Lösung, die aber auf ein deutlich komplizierteres Programm führt und deren Korrektheit alles andere als offensichtlich ist).

Im Allgemeinen ermöglicht Rekursion oft elegantere Lösungen als sie ohne Rekursion möglich wären. Wir werden weitere Anwendungen der Rekursion im zweiten Semester kennenlernen.

1.8 Zeiger (Pointer) und Abstrakte Datentypen (ADT)

Ein Datentyp stellt im Allgemeinen einen Wertebereich dar (dies sind die Werte, die eine Variable des Datentyps annehmen kann) und hat eine zugehörige Menge an Operatoren (Funktionen), die Variablen dieses Datentyps als formale Parameter (oder Rückgabeparameter) haben. Bei vordefinierten Datentypen sind durch die Programmiersprache der Wertebereich und die Operatoren festgelegt. Der Datentyp `int` hat so in der Regel den Wertebereich -2^{31} bis $2^{31}-1$ und als Operatoren die Grundrechenarten, Vergleichsoperatoren und einige weitere, die wir hier nicht im Detail weiter besprechen. Formal sind die Operatoren Funktionen, z. B. ist „+“

eine Funktion mit Definitionsbereich `int`×`int` und Zielbereich `int`. Die Implementierung ist durch den Compiler und die Hardware festgelegt.

1.8.1 Die ADTs Stack und Queue

Mitte der 1970er Jahre wurde der Begriff des Abstrakten Datentyps (ADT) eingeführt. Darunter versteht man die Spezifikation eines Datentyps unabhängig von einer konkreten Implementierung. Wir führen das Konzept an zwei Beispielen ein: dem Datentyp „Stack“ (ein Stapelspeicher) und dem Datentyp „Queue“ (eine Warteschlange).

Ein Stack arbeitet nach dem Prinzip „Last-in-First-out“ (LIFO) – analog zum Füllen eines Umzugskartons: das, was zuletzt oben hinzugefügt wurde, wird beim Auspacken als erstes wieder entnommen. Um solche Prozesse beschreiben zu können, müssen wir uns überlegen, welche Operatoren wir benötigen, um mit einem Stack arbeiten zu können:

- Eine Funktion `empty` – diese soll einen leeren Stack erzeugen – formal eine Funktion ohne Eingangsparameter und mit einem Ausgangsparameter vom Datentyp `stack` – somit `empty : → stack`
- Eine Funktion `is_empty` – wenn später Elemente vom Stapel genommen werden sollen, muss vorher überprüfbar sein, ob sich überhaupt noch Elemente auf diesem befinden; als Ergebnis erhält man einen Wahrheitswert – `is_empty : stack → bool`
- Eine Funktion `push` – um auf einen bereits vorhandenen Stapel ein weiteres Element ablegen zu können, das Ergebnis der Funktion ist wieder ein Stack – `push : stack × element → stack`
- Eine Funktion `pop` – um das oberste Element von einem Stack zu entfernen – `pop : stack → stack`
- Eine Funktion `top` – um das oberste Element von einem Stack zu lesen (der Stack bleibt dabei unverändert) – `top : stack → element`

Diese formale Angabe der Eingangs- und Ausgangsparameter einer Funktion nennt man auch Signatur (gewissermaßen die Syntax der Funktion).

Wir wollen uns hier auf die obige Funktionalität beschränken – insbesondere gibt es keine Möglichkeit auf weiter unten liegende Elemente direkt zuzugreifen oder weiter unten liegende Elemente zu entfernen, ohne die darüberliegenden zu verändern.

Betrachten wir im Vergleich die Warteschlange:

Eine Queue arbeitet nach dem Prinzip „First-in-First-out“ (FIFO) – analog zum Anstellen in der Mensa, wer zuerst kommt, bekommt auch zuerst sein Essen (das Vordrängeln wird hier nicht modelliert). Die Operatoren einer Warteschlange sind ähnlich wie die eines Stacks:

- Eine Funktion `empty` – diese soll eine leere Queue erzeugen – `empty : → queue`
- Eine Funktion `is_empty` – analog zum Stack – `is_empty : queue → bool`
- Eine Funktion `enqueue` – um an eine bereits vorhandene Warteschlange ein weiteres Element anfügen zu können – `enqueue : queue × element → queue`

- Eine Funktion `dequeue` – um das vorderste Element von einer Queue zu entfernen – `dequeue : queue → queue`
- Eine Funktion `front` – um das vorderste Element von einer Queue zu lesen (ohne es zu entfernen) – `front : queue → element`

Wenn wir uns die Signaturen der Warteschlangenoperatoren anschauen, so stellen wir fest, dass diese aus einer reinen Umbenneung des Stacks hervorgehen (`stack` heißt jetzt `queue`, `push` heißt jetzt `enqueue`, `pop` heißt jetzt `dequeue`, `top` heißt jetzt `front`). Ein ADT ist also nicht allein durch seine Signaturen eindeutig beschrieben. Wir müssen die Bedeutung (die Semantik) der Funktionen noch genauer spezifizieren. Hierzu sind die beiden üblichsten Methoden die „informelle Methode“ und die „mathematisch-axiomatische Methode“.

Die informelle Methode eignet sich insbesondere, um diese Information später als Kommentarzeilen im Code zu übernehmen, für jede Funktion wird dabei angegeben,

- welche Vorbedingungen erfüllt sein müssen – wenn z. B. die Funktion `pop` aufgerufen wird, darf der übergebene Stack nicht leer sein
- welchen Effekt / welches Ergebnis die Funktion hat – bei der Funktion `push` wäre dies, dass danach auf dem Stack das übergebene Element oben liegt (und sich darunter die Reihenfolge der Elemente im Stack nicht verändert hat)
- ggf. noch, was die Funktion tut, wenn die Vorbedingung nicht erfüllt ist – dies beeinflusst die spätere Implementierung; man könnte das Verhalten entweder undefiniert lassen und somit dem Benutzer die Verantwortung überlassen, dass z. B. die Funktion `top` nur aufgerufen wird, wenn der Stack nicht leer ist, oder definieren, dass eine Fehlermeldung auf der Konsole ausgegeben wird (und der Stack unverändert bleibt), oder fordern, dass über eine weitere boolesche Variable solche Fehlermeldungen mitgeteilt werden (wobei letztere Variante dann auch eine Änderung der Signaturen nach sich ziehen würde)

– solche Informationen sollten (unabhängig vom Konzept ADT) immer in den Kommentarzeilen um den Funktionskopf vorhanden sein.

Da die obige Beschreibung in der Regel umgangssprachlich ist, kann sie nicht direkt dazu verwendet werden, um die korrekte Funktionsweise des Stacks zu überprüfen. Bei der mathematisch-axiomatischen Methode geben wir stattdessen Eigenschaften in Regelform an, die der Datentyp erfüllen soll. Für den ADT Stack könnte man z. B. folgende Regeln angeben (Achtung, dies ist noch kein C/C++-Code) für einen beliebigen Stack `s` und ein beliebiges Element `x`:

- `is_empty(empty()) = TRUE`
- `is_empty(push(s,x)) = FALSE`
- `pop(s)` → `ERROR`, falls `s` leer ist
- `pop(push(s,x)) = s`
- `top(push(s,x)) = x`
- `push(pop(s),top(s)) = s`, falls `s` nicht leer ist (hier wird davon ausgegangen, dass die Parameter jeweils auf Basis des Ausgangsstacks ausgewertet werden – dies ist in der konkreten Programmiersprache nicht unbedingt garantiert)

Wenn man die Implementierung eines Stacks vorliegen hat, kann man mit Hilfe dieser Regeln die korrekte Implementierung testen.

Angestrebte Eigenschaften eines ADTs: Die Frage bleibt, ob mit der nun so angegebenen Semantik, der ADT eindeutig beschrieben ist (überprüfen Sie selbst, welche der Regeln mit der oben angegebenen Umbennung auch für eine Queue gelten würden). Wir geben hier einige Eigenschaften an, die im Allgemeinen bei ADTs angestrebt werden.

- Universalität – einfache (Wieder-)Verwendung eines ADT in jedem Programm, ohne den ADT auf die konkrete Anwendung anpassen zu müssen
- Präzise Beschreibung – die Schnittstelle zwischen Implementierung und Anwendung muss eindeutig und vollständig sein (bei Verwendung der mathematisch-axiomatischen Methode müssen so z. B. Regeln vorhanden sein, die nicht sowohl für einen Stack als auch für eine Queue gelten)
- Einfachheit – ein ADT kann ohne Wissen über die innere Realisierung verwendet werden, insbesondere übernimmt ein ADT seine Repräsentation und Verwaltung im Speicher selbst
- Kapselung – dem Anwender soll durch die Schnittstelle sehr genau wissen, was ein ADT tut, aber es soll verborgen sein, wie er es tut
- Geschütztheit – der Anwender soll keine Zugriffsmöglichkeit auf die interne Struktur der Daten haben; der Zugriff auf die Daten soll außschließlich über die definierte Schnittstelle eines ADT geschehen können
- Modularität – dient zum Einen dazu, den so in sich abgeschlossene Teil eines Programms unabhängig von anderen Modulen zu realisieren und zu testen; zum Anderen erlaubt es einen leichten Austausch von Programmteilen, wenn effizientere Methoden gefunden werden, Verbesserungen sollen so in ADTs nachträglich möglich sein, ohne Änderungen in den das Modul verwendenen Programmteilen vornehmen zu müssen

Die Anforderungen sind nicht immer 100%-ig zu erfüllen, sollten aber als Ziel dienen, um so Datenstrukturen und die darauf basierenden Funktionen möglichst flexibel einsetzen zu können und sie gleichzeitig wartbar zu halten. Manche Programmierparadigmen, wie z. B. die objektorientierte Programmierung, unterstützen die Einhaltung obiger Ziele automatisch. Im klassischen Kern von C/C++ sind Sie als Programmierer stärker gefordert, auf diese Eigenschaften zu achten.

1.8.2 Dynamische Variablen in C/C++ mit `new` und `delete` (oder `malloc` bzw. `free`)

Zurück zur Umsetzung des ADTs Stack in C/C++: Betrachten wir zunächst die Umsetzung der Signaturen mit Verwendung des Rückgabekonzepts – wir gehen davon aus, dass wir einen Datentyp `DataType` für die im Stack abzulegenden Elemente haben, der Datentyp für den Stack heiße `StackPtr`

```
StackPtr empty(void) { ... }
bool is_empty(StackPtr ...) { ... }
StackPtr push(StackPtr ..., DataType ...) { ... }
StackPtr pop(StackPtr ...) { ... }
DataType top(StackPtr ...) { ... }
```

Im bisherigen strukturierten Entwurf hätten wir erkannt, dass die Ein- und Ausgangsparameter bei `push` und `pop` in Wahrheit Durchgangsparameter sind, und wir hätten so die folgenden Funktionen erhalten:

```
empty(StackPtr & ...) { ... }
is_empty(const StackPtr ..., bool & ...) { ... }
push(StackPtr & ..., const DataType ...) { ... }
pop(StackPtr & ...) { ... }
top(const StackPtr ..., DataType & ...) { ... }
```

In der Regel können wir vorher nicht wissen, wieviele Elemente maximal auf den Stack gelegt werden sollen. Die Möglichkeit einen Stack als Array zu implementieren scheidet also aus (oder würde das angestrebte Ziel der Universalität einschränken). Wir müssten während der Laufzeit in der Lage sein, neue Variablen des `DataType`s anzulegen und im Sinne des Stacks zu verknüpfen.

C++ stellt hierfür die Funktion `new` zur Verfügung – diese lässt sich universell einsetzen, um während der Laufzeit Speicherplatz für Variablen eines Datentyps anzulegen. Das Ergebnis der Funktion ist dann die Speicheradresse, an der der angeforderte Speicherplatz beginnt. Um sich diese Speicheradresse merken zu können, benötigen wir geeignete Variablen, die Zeiger genannt werden (der englische Begriff Pointer ist im deutschen Sprachraum ebenfalls üblich). Zur Deklaration einer Zeiger-Variablen fügen wir dem Datentyp ein „`*`“ an.[4]

Wie können wir nun mit `new` geeignet Speicherbereich reservieren und damit einen Stack implementieren? Zur Erinnerung: ein Stack hat nur direkten Zugriff auf sein oberstes Element. Wird dieses entfernt, muss das oberste Element wissen, welches Element das nächste im Stapel ist. Jedes Element hat also einen Wert und eine Information, an welcher Speicheradresse sich das nächste Element des Stapels befindet (die Angabe einer Speicheradresse erfolgt in einem Zeiger). Da hier stets nur mit Verweisen auf nächste Elemente gearbeitet wird, erfolgt auch der Zugriff auf das oberste Element mit solch einem Zeiger. Die Datenstruktur (Wert + Speicheradresse) lässt sich mit einem Verbund beschreiben:

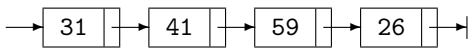
```
struct Stack {
  DataType value;
  Stack * next; // Zeiger auf das nächste Element im Stapel
} ;
```

[4] C/C++ ist hier leider etwas verwirrend – genau genommen gehört der „`*`“ zum Variablennamen! Die Deklaration `int* a1, a2;` erzeugt so einen Zeiger `a1` auf eine `int`-Variable und eine `int`-Variable `a2`. Will man zwei Zeiger-Variablen, hätte man `int *a1, *a2;` schreiben müssen – die Position der Leerzeichen spielt dabei keine Rolle! Definiert man sich einen `typedef int* intptr;`, so sind mit `intptr a1,a2;` (erwartungsgemäß) beides Zeiger-Variablen.

Prinzipiell kann man mit diesem Datentyp bereits arbeiten[5]. Um speziell zu Beginn des Arbeitens mit Zeigern die Verwirrung durch die Markierung mit dem „*“ geringer zu halten, definieren wir unsere Datenstruktur etwas anders und führen einen eigenen Typ für den Pointer ein:

```
struct Stack; // nur eine Vorabbekanntmachung, damit der
typedef Stack * StackPtr; // separate Pointer-Typ definiert werden kann
struct Stack {
  DataType value;
  StackPtr next;
};
```

Damit sind wir – zumindest vorübergehend – den „*“ wieder los. Zur Veranschaulichung solch einer Datenstruktur bedient man sich häufig folgender grafischer Darstellung (stellen Sie sich den Stapel nach links auf die Seite gekippt vor) – wir nehmen hier an, dass wir Zahlen im Stapel gespeichert haben:



Obiges entspricht einem Stapel, auf den zuerst das Element `26` gelegt wurde, danach die Elemente `59`, `41` und `31`, so dass die `31` jetzt das oberste Element ist. Der senkrechte Strich markiert das Ende des Stacks – in C/C++ wird dafür die spezielle Speicherstelle 0 benutzt. Damit sind die Funktionen für einen Stack leicht zu implementieren:

- `empty` muss lediglich dafür sorgen, dass der Zeiger auf den Stack auf die spezielle Speicherstelle 0 zeigt (im C++-Code, der im Abschnitt 1.8.5 zu finden ist, ist die Funktion etwas anders implementiert, so dass der Speicher, der von einem vorher über diese Variable verwalteten Stack verwendet wurde, zunächst wieder freigegeben wird – siehe auch die Anmerkungen zur Funktion `pop` ein paar Zeilen weiter unten)
- `is_empty` überprüft, ob der Zeiger auf den Stack auf 0 zeigt
- `push` erzeugt ein neues , das an den Anfang des Stacks gestellt wird (der Zeiger auf das neue oberste Element zeigt dann auf dieses neu erzeugte Element; der Zeiger auf dessen nächstes Element zeigt auf das bisherige oberste Element (oder auf 0, falls der Stack vorher leer war))
- `pop` entfernt das oberste Element – der Zeiger zeigt danach nicht mehr auf das bisherige oberste Element, sondern auf dessen Nachfolger (im obigen Beispiel würde das neue oberste Element auf die `41` zeigen); danach existiert kein Zugriff mehr auf das bisherige oberste Element, der verwendete Speicher kann (und sollte) wieder freigegeben werden (dies übernimmt der Befehl `delete` – die Anwendung ist dem Code in Abschnitt 1.8.5 zu entnehmen)
- `top` liest das oberste Element aus. Der Zeiger darf – wie auch bei `pop` – dabei nicht auf die Speicherstelle 0 zeigen; bei einem versuchten Lesezugriff auf diese Speicherstelle bricht das Betriebssystem das Programm ab

[5] Hinweis: obiges funktioniert nur in C++ – in C hätte die entsprechende Zeile `struct Stack * next;` heißen müssen.

Der Zugriff auf Elemente in Verbunden unterscheidet sich beim Zugriff über Pointer von dem über direkten Zugriff. Bei letzterem hatten wir mit der Punkt-Notation gearbeitet, um auf die Komponenten des Verbundes zuzugreifen. Hat man nur einen Pointer auf den Verbund, so muss man mit der Pfeil-Notation arbeiten (d. h., einfach den „`.`“ durch ein „`->`“ ersetzen). Auch hier sei auf den Abschnitt 1.8.5 verwiesen.

Beschränkt man sich auf den Sprachanteil in C, so muss man einige zusätzliche Feinheiten beachten. Der Compiler sorgt nicht mehr automatisch für die Größe des zu reservierenden Speicherbereichs – der Programmierer muss diese ggf. selbst z. B. mit der Funktion `sizeof()` bestimmen und kann dann mit `malloc()` den Speicher anfordern. Das Freigeben geschieht dann mittels `free()`. Wir gehen auf die Details hier nicht weiter ein, da wir in unseren Programmen bereits nur in C++ verfügbare Sprachanteile benutzen (z. B. Referenzparameter) und so auch hier die leichter zu handhabenen Funktionen `new` und `delete` für die dynamische Speicherverwaltung verwenden.

Wir betrachten die Implementierung des ADT Queue nach der Einführung des ADT Liste im nun folgenden Abschnitt.

1.8.3 Der ADT Liste

Arrays erlauben uns zwar auf viele gleichartige Datentypen über einen Bezeichner (und Index) zuzugreifen – die (maximale) Anzahl der Elemente muss jedoch vorab bekannt sein. Der abstrakte Datentyp Liste schafft hier Abhilfe und erlaubt, dynamisch Elemente zu einer Sammlung gleichartiger Daten hinzuzufügen und zu löschen. Das heißt insbesondere, dass wir in der Lage sein müssen, während der Laufzeit Speicherplatz für das Programm anzufordern und diesen später wieder freizugeben. Wir haben die notwendigen Befehle bereits beim Stack eingeführt.

Es spricht nichts dagegen, einfach die Typdefinition des Stacks zu verwenden und die Menge der auf diesem Typ arbeitenden Funktionen zu erweitern. Wie beim Stack wird man einen Zeiger auf das erste Element der Liste haben (bei Listen spricht man dabei vom *Anker*). Beim Einfügen von Elementen ist man nun aber nicht mehr darauf beschränkt, dieses an den Anfang der Liste zu stellen – es bieten sich hier viele Varianten an:

- Einfügen am Beginn der Liste (sehr einfach, wie beim Stack)
- Einfügen am Ende der Liste (dazu muss man mit einem weiteren Hilfszeiger sich über die `next`-Verweise bis zum bisher letzten Element vorarbeiten, um dann das neue als dessen Nachfolger einzufügen – beachte, die Liste könnte vorher leer sein (überlegen Sie selbst, warum es beim `push`-Aufruf keine Rolle gespielt hat, ob der Stack vorher leer war oder nicht)
- sortiertes Einfügen – dazu muss die Liste, in die eingefügt wird, natürlich vorher schon sortiert sein; in diesem Fall können mehrere Situationen auftreten:
  - die Liste war vorher leer
  - das neue Element ist danach das kleinste der Liste (d. h., der Anker wird sich ändern)
  - das neue Element muss zwischen zwei bestehenden Elementen der Liste eingefügt werden
  - das neue Element ist danach das größte der Liste, wird also am Ende neu eingefügt

Analog zum Einfügen kann man sich bei Listen abhängig von der geplanten Anwendung viele verschiedene Funktionen ausdenken, die sinnvoll sein könnten, als Anregung – ohne Anspruch auf Vollständigkeit – seien folgende Funktionen genannt:[6]

- Ausgabe der Elemente in der Liste
- Funktionen zum Suchen:
  - Überprüfe, ob ein gegebenes Element in der Liste vorkommt
  - Überprüfe, ob es Elemente gibt, die doppelt vorkommen
  - Finde das kleinste/größte Element
- Funktionen zum Bearbeiten von Listen:
  - Sortieren der Elemente in der Liste
  - Entfernen des ersten/letzten Elements
  - Entfernen des ersten Elements mit einem bestimmten Wert
  - Entfernen aller Elemente mit einem bestimmten Wert
  - Entfernen aller doppelten Elemente
- ... lassen Sie Ihrer Fantasie freien Lauf ...

Die im Abschnitt 1.8.5 für Listen angegebenen Funktionen sind lediglich umbenannte Kopien von denen vom Stack. Als Übungsaufgabe sollten Sie das sortierte Einfügen in eine Liste implementieren und sich mindestens 5 weitere Funktionen ausdenken (oder aus obiger Aufzählung wählen) und diese erfolgreich umsetzen.

1.8.4 Varianten von Listen oder der ADT Queue

Bei der Überlegung, wie man Queues implementieren könnte, entdeckt man weitere sinnvolle Varianten von Listen.

- Wenn man lediglich die Datenstruktur des Stacks kopiert (und bei `enqueue` wie in der Funktion `push` das neue Element an den Anfang der Liste stellt), werden die Funktionen `dequeue` und `front` sehr aufwendig, da man jedesmal die gesamte Liste durchlaufen muss, um das betreffende Element zu finden. Verwaltet man zusätzlich zum Zeiger auf das letzte Element einen Zeiger auf das vorderste Element (also das, auf das man mit der Funktion `front` zugreifen möchte), so ist zwar vorübergehend der Zugriff auf `front` beschleunigt, nach dem nächsten `dequeue` ist der Zeiger aber nicht mehr gültig und der Zeiger auf das nun vorderste Element ist nur aufwendig zu bestimmen.
- Zur Lösung könnte man zusätzlich zu den `next`-Zeigern in der Datenstruktur einen Zeiger auf das „links“ davon stehende Element einführen. Im `struct` gäbe es dann zwei Zeiger, einen auf das links folgende, einen auf das rechts folgende Element. Zusammen mit der

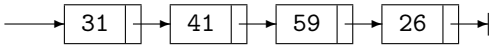
[6] Während bei Stacks die Funktionsnamen `push`, `pop` und `top` in der Literatur sehr einheitlich verwendet werden, hat sich bei Listen keine einheitliche Namensgebung durchsetzen können – geben Sie den Funktionen einfach sprechende Namen

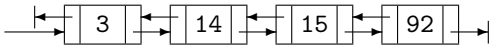
Idee, sich zwei Zeiger auf das erste und letzte Element zu merken, sind nun alle Funktionen schnell durchzuführen. Gegenüber dem Stack bleibt jedoch noch der Nachteil, dass doppelt soviel Speicher für die Zeiger verwendet wurde und so innerhalb der Funktionen auch mehr Zeiger verändert werden müssen.

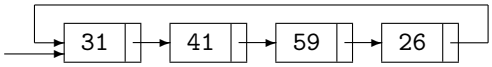
- Bei näherer Betrachtung dieser doppelt verketteten Liste fällt auf, dass wir lediglich die Zeiger auf das „links“ stehende Element verwendet haben – dies reduziert den Speicherplatzverbrauch zwar auf das beim Stack verwendete Maß, unschön bleibt aber, dass wir weiterhin zwei Zeiger auf das erste und letzte Element der Queue verwalten müssen.
- Beachte: der „links“-Zeiger des letzten Elements der Queue wird nicht verwendet: Wir greifen auf das letzte Element der Queue direkt zu und müssen das Ende der Queue nicht über den Null-Zeiger (der auf die Speicheradresse 0 zeigt) erkennen. Die Lösung: statt sich den Zeiger auf das erste Element der Queue separat zu merken, speichern wir diesen im „links“-Zeiger des letzten Elements der Queue. Dadurch bilden die Elemente der Queue eine Ringliste!

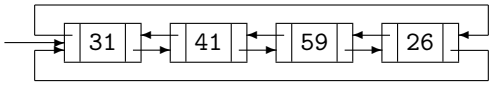
Der C++-Code für die elegante Ringlisten-Lösung ist dem folgenden Abschnitt 1.8.5 zu entnehmen.

Übersicht über Varianten von Listen:

- Einfach verkettete Listen – wir merken uns neben dem Inhalt nur einen Zeiger auf das „rechts“ davon stehende Element: Verwendung typischerweise bei Stacks und bei den meisten Listen-Anwendungen 

- Doppelt verkettete Listen – zu jedem Element merken wir uns Zeiger auf das „links“ und „rechts“ davon stehende Element: Verwendung typischerweise bei Verwaltung sortierter Daten, bei denen ein Zugriff auf Vorgänger und Nachfolger in der sortierten Folge notwendig ist 

- Einfach verkettete Ringlisten – wie bei einfach verketteten Listen, der „rechts“-Zeiger des letzten Elements verweist aber wieder auf das erste Element: Verwendung typischerweise beim ADT Queue und bei Anwendung von Puffern 

- Doppelt verkettete Ringlisten – na was wohl?: wird selten benutzt, die typischen Anwendungsfälle benutzen einen der anderen drei Typen 

Bei beiden Ringlisten-Varianten kann der Anker prinzipiell auf ein beliebiges Element zeigen (es gibt kein ausgezeichnetes erstes Element mehr).

1.8.5 C/C++-Code der Module stack, queue und liste

Zum Abschluss des Abschnitts über Zeiger und ADTs folgt noch der C/C++-Code der Beispiel-Module und ein Beispiel, wie diese Module eingebunden und verwendet werden können.

Man beachte, wie zur Vermeidung der Mehrfachdefinition des Datentyps `DataType` das Konzept wie beim allgemeinen Einbinden der Module verwendet wird.

Die Datei `stack.h`

```
#ifndef STACK_H
#define STACK_H

#ifndef DataTypeDefined
#define DataTypeDefined
typedef int DataType;
#endif

struct Stack;
typedef Stack * StackPtr;

void empty(StackPtr &);                 // leert einen Stack
bool is_empty(const StackPtr);          // prüft, ob Stack leer ist
void push(StackPtr &, const DataType);  // legt Data auf Stack ab
void pop(StackPtr &);                   // entfernt oberstes Element vom Stack
DataType top(const StackPtr);           // liest oberstes Element vom Stack

#endif
```

Die Datei `stack.cpp`

```
#include "stack.h"

struct Stack {
  DataType value;
  StackPtr next;
};

void empty(StackPtr & s) {
  while (!is_empty(s)) { pop(s); }
  // s = 0; würde auch funktionieren, Speicher würde aber nicht freigegeben
}

bool is_empty(const StackPtr s) {
  return s==0; // Kurzform von if (s==0) { return true; } else { return false; }
}

void push(StackPtr & s, const DataType value) {
  StackPtr node = new Stack;
  node->value = value;
  node->next = s;
  s = node;
}

void pop(StackPtr & s) { // s darf nicht 0 sein!
  // die aufrufende Funktion muss ggf. mit is_empty(s) prüfen, ob s leer ist
```

```
  StackPtr tbd = s; // Zeiger für oberstes Element merken
  s = s->next;
  delete tbd;
}

DataType top(const StackPtr s) { // s darf nicht 0 sein!
  // die aufrufende Funktion muss ggf. mit is_empty(s) prüfen, ob s leer ist
  return s->value;
}
```

Die Datei `queue.h`

```
#ifndef QUEUE_H
#define QUEUE_H

#ifndef DataTypeDefined
#define DataTypeDefined
typedef int DataType;
#endif

struct Queue;
typedef Queue * QueuePtr;

void empty(QueuePtr &);                    // leert eine Queue
bool is_empty(const QueuePtr);             // prüft, ob Queue leer ist
void enqueue(QueuePtr &, const DataType);  // fügt Data an Queue an
void dequeue(QueuePtr &);                  // entfernt erstes Element aus Queue
DataType front(const QueuePtr);            // liest erstes Element aus Queue

#endif
```

Die Datei `queue.cpp`

```
#include "queue.h"

struct Queue { // einfach verkettete Ringliste: Anker zeigt auf das Queue-Ende
  DataType value; // front zeigt allgemein auf das nachfolgende Element
  QueuePtr front;  // "Anker"->front zeigt als Ringschluss auf das vorderste
};

void empty(QueuePtr & q) {
  while (!is_empty(q)) { dequeue(q); }
  // q = 0; würde auch funktionieren, Speicher würde aber nicht freigegeben
}

bool is_empty(const QueuePtr q) {
  return q==0; // Kurzform von if (q==0) { return true; } else { return false; }
}
```

```
void enqueue(QueuePtr & q, const DataType value) {
  if (q==0) {
    q = new Queue; // erstes Element in der Queue
    q->value=value;
    q->front=q; // erstes und letztes Element sind identisch
  } else {
    QueuePtr node = new Queue;
    node->value = value;
    node->front = q->front;
    q->front = node;
    q = node;
  }
}

void dequeue(QueuePtr & q) { // q darf nicht 0 sein!
  // die aufrufende Funktion muss ggf. mit is_empty(q) prüfen, ob q leer ist
  if (q->front==q) { // ist es das einzige Element in der Queue?
    delete q;
    q=0;
  } else {
    QueuePtr tbd = q->front; // Zeiger für zu löschendes Element merken
    q->front = q->front->front;
    delete tbd;
  }
}

DataType front(const QueuePtr q) { // q darf nicht 0 sein!
  // die aufrufende Funktion muss ggf. mit is_empty(q) prüfen, ob q leer ist
  return q->front->value;
}
```

Die Datei `liste.h`

```
#ifndef LISTE_H
#define LISTE_H

#ifndef DataTypeDefined
#define DataTypeDefined
typedef int DataType;
#endif

struct Liste;
typedef Liste * ListenPtr;

void empty(ListenPtr &);                 // leert eine Liste
bool is_empty(const ListenPtr);          // prüft, ob Liste leer ist
void insert(ListenPtr &, const DataType); // fügt Data sortiert in Liste ein
```

```
void removeFirst(ListenPtr &);          // entfernt erstes Element der Liste
DataType head(const ListenPtr);         // liest erstes Element der Liste

#endif
```

Die Datei `liste.cpp`

```
#include "liste.h"

struct Liste {
  DataType value;
  ListenPtr next;
};

void empty(ListenPtr & l) {
  while (!is_empty(l)) { removeFirst(l); }
  // l = 0; würde auch funktionieren, Speicher würde aber nicht freigegeben
}

bool is_empty(const ListenPtr l) {
  return l==0; // Kurzform von if (l==0) { return true; } else { return false; }
}

void insert(ListenPtr & l, const DataType value) { // Aufgabe:
  // ändern Sie die Funktion so ab, dass <value> sortiert in <l> eingefügt wird!
  ListenPtr node = new Liste;
  node->value = value;
  node->next = l;
  l = node;
}

void removeFirst(ListenPtr & l) { // l darf nicht 0 sein!
  // die aufrufende Funktion muss ggf. mit is_empty(l) prüfen, ob l leer ist
  ListenPtr tbd = l; // Zeiger für oberstes Element merken
  l = l->next;
  delete tbd;
}

DataType head(const ListenPtr l) { // l darf nicht 0 sein!
  // die aufrufende Funktion muss ggf. mit is_empty(l) prüfen, ob l leer ist
  return l->value;
}
```

Das Hauptprogramm `int main()` Dieses liest solange Zahlen ein und fügt diese in einen Stack, eine Queue und eine sortierte einfach verkettete Liste ein bis der Benutzer die Zahl 0 eingibt. Danach werden die Inhalte der drei Datenstrukturen wieder ausgegeben (und der verwendete Speicher wieder freigegeben). In der Anwendung sollte man Zeiger immer mit `0` initialisieren, ggf. ist ein kontrollierter Abbruch eines Programms durch das Betriebssystem alle

Mal besser, als wenn ein Zeiger mit einem zufälligen Wert in einen zum Programm gehörenden Speicherbereich zeigt und dort unkontrolliert Daten verändert.

Man sollte hier auch beachten, dass man die Schnittstelle einer Datenstruktur benutzt. Die zusätzlichen Aufrufe von `empty` für die drei Datenstrukturen scheinen überflüssig, da zuvor schon die Zeiger mit `0` initialisiert wurden. Die Schnittstellen sind jedoch gerade dazu da, dass man nur über diese auf die Datenstruktur zugreift und diese möglichst ohne vermeintliches Wissen über die interne Darstellung verwendet. Stellen Sie sich vor, eines der Module würde so verändert, dass statt der Zeiger-Implementierung ein ausreichend großes Array vorliegt (analog zum Datentyp `Menge` in Abschnitt 1.6.6). Aufgrund der fehlenden Initialisierung mit `empty` würde dynamisch kein entsprechend großes Array angelegt – der erste Zugriff auf die Datenstruktur (egal welcher Art) würde einen Programmabsturz zur Folge haben.

```
#include <iostream>  // für die einfache Textausgabe

#include "stack.h"
#include "queue.h"
#include "liste.h"

using namespace std; // für iostream

int main(void) {
  cout << "Zahleneingabe bis 0; Einfuegen in Stack, Queue und sortiert in Liste." << endl;

  StackPtr s=0; QueuePtr q=0; ListenPtr l=0;
  // Zeiger immer mit 0 initialisieren!
  empty(s); empty(q); empty(l);
  DataType zahl;

  cout << "Erste Zahl: "; cin >> zahl;
  while (zahl) { // Kurzform für while (zahl!=0)
    push(s,zahl); enqueue(q,zahl); insert(l,zahl);
    cout << "Nexte Zahl: "; cin >> zahl;
  }
  cout << "Ausgabe vom Stack (inkl. Loeschen der Elemente): ";
  while (!is_empty(s)) {
    cout << top(s) << " ";
    pop(s);
  }
  cout << endl;
  cout << "Ausgabe der Queue (inkl. Loeschen der Elemente): ";
  while (!is_empty(q)) {
    cout << front(q) << " ";
    dequeue(q);
  }
  cout << endl;
  cout << "Ausgabe der Liste (inkl. Loeschen der Elemente): ";
  while (!is_empty(l)) {
    cout << head(l) << " ";
```

```
    removeFirst(l);
  }
  cout << endl;
  return 0;
}
```

2 Endliche Automaten

Als kurze Einführung in endliche Automaten sei hier für den Moment lediglich auf die Seite bei Wikipedia verwiesen: `http://de.wikipedia.org/wiki/Endlicher_Automat`

Endliche Automaten bieten gegenüber der allgemeinen Programmierung zwar nur eingeschränkte Modellierungsmöglichkeiten, sind dadurch aber auch leicht verständlich und lassen sich ebenfalls automatisch in C/C++-Programme umsetzen. Oft lässt sich die Grobstuktur eines Programms gut und leicht mit Endlichen Automaten verdeutlichen, ohne dass man sich in dem Moment genau überlegen muss, wie dies später in ein C/C++-Programm umzusetzen ist.

A Kompilieren der Übungsbeispiele

Als Programmieranfänger hat man gleichzeitig an zwei Fronten zu kämpfen: die Programmiersprache ist neu und nicht minder fordernd ist die Entwicklungsumgebung. Die folgenden Schritt-für-Schritt-Anleitungen sollen helfen, einen Einstieg in verschiedene Entwicklungsumgebungen zu finden.

A.1 Microsoft Visual Studio

Im Labor wird die Entwicklungsumgebung MS Visual Studio C/C++ verwendet. Um den Einstieg zu Erleichtern geben wir hier zunächst an, wie ein neues Projekt erzeugt wird. Sie sollten sich in Ihrem Home-Verzeichnis ein Verzeichnis für das Labor anlegen, in dem die Entwicklungsumgebung dann für die einzelnen Programme einzelne Verzeichnisse erzeugen wird. Hier ist Version 6.0 verwendet worden, das Vorgehen sollte sich bei aktuelleren Versionen jedoch nicht wesentlich unterscheiden.

1. Starten Sie Microsoft Visual Studio
2. Wählen Sie im Menü `File`→`New`
3. Im neu erscheinenden Fenster wählen Sie im Reiter `Projects` (ist vorgewählt) links `Win32 Console Application`, rechts als `Location` Ihr für das Labor angelegte Verzeichnis sowie danach einen passenden `Project name` (der Speicherort verändert sich simultan mit). Durch den auch vorgewählten Punkt `Create new workspace` wird für das Projekt ein eigenes Verzeichnis innerhalb Ihres Labor-Verzeichnisses automatisch angelegt. → `OK`
4. Wählen Sie im dann erscheinenden Fenster stets `An empty Project.` → `Finish` → `OK`
5. Falls nicht voreingestellt, wählen Sie in dem nun erscheinenden Editor links `File View` (statt `Class View`)

Bei bereits vorliegenden Quelldateien können Sie diese nun im `File View` über das Kontextmenü (rechte Maustaste) hinzufügen (`Add Files to Folder` – Dateien mit Endung `.cpp` im Verzeichnis Source, Dateien mit Endung `.h` im Verzeichnis Header). Das Programm kann nun über `Build`→`Build <project_name>.exe` (F7) übersetzt und dann über `Build`→`Execute <project_name>.exe` (je nach Tastatur mit Strg-F5 oder Ctrl-F5) gestartet werden.

Liegen noch keine Quell-Dateien vor, können Sie diese über `File`→`New` im Reiter `Files` neue `C++ Source File`s oder `C/C++ Header File`s erstellen und auch automatisch Ihrem Projekt zuweisen (`Add to project`). Das Übersetzen und Starten funktioniert dann wie oben.

A.2 MS Visual Studio C++ 2005 / 2008 Express

Diese frei verfügbare Version ist eine abgespeckte Variante der Vollversion von MS Visual Studio (von Version 8 (2005) bzw. Version 9 (2008)) und wird mittlerweile auch im Labor benutzt. Die Schritte sollten sich in anderen Versionen nur unwesentlich unterscheiden:

1. Alle Dateien eines Projekts in ein ansonsten leeres Verzeichnis Ihrer Wahl abspeichern
2. Microsoft Visual C++ 2005 / 2008 Express starten
3. Menüauswahl Datei→Neu→Projekt – im sich öffnenden Fenster „Neues Projekt“
   - links Win32 wählen
   - rechts Win32-Konsolenanwendung
   - Name: passenden Namen wählen
   - Speicherort: das Verzeichnis wählen, in das im Schritt 1 die Dateien abgespeichert wurden
   - Projektmappe: Häkchen bei Projektmappenverzeichnis erstellen entfernen
   - „Ok“ klicken
4. Im sich öffnenden Fenster „Win32-Anwendungs-Assistent“ links die „Anwendungseinstellungen“ anklicken
   - als Anwendungstyp „Konsolenanwendung“ wählen
   - das Häkchen bei „Vorkompilierte Header“ entfernen
   - das Häkchen bei „Leeres Projekt“ setzen
   - „Fertig stellen“ klicken
5. Menüauswahl Ansicht→Projektmappen-Explorer – dort sollte sich nun eine Projektmappe mit dem in Schritt 3 gewählten Namen befinden
6. ggf. das „+“ vor dem Projektmappen-Namen klicken; es erscheinen die Unterpunkte „Headerdateien“, „Quelldateien“ und „Ressourcendateien“
   - Im Kontextmenü (rechte Maustaste) von Quelldateien den Menüpunkt Hinzufügen→Vorhandenes Element wählen
   - Im sich öffnenden Dateidialog das Verzeichnis aus Schritt 1 wählen (in der Regel ist dies eine Ebene über dem aktuellen), dort alle Dateien mit Endung `.cpp` auswählen und „Hinzufügen“ wählen

- Diesen Schritt im Kontextmenü von Headerdateien mit den Headerdateien (Endung `.h`) wiederholen

7. ggf. Doppelklick im Projektmappen-Explorer auf eine Datei; diese öffnet sich im Editor
8. Menüauswahl Erstellen→Projektmappe erstellen – das Programm wird (hoffentlich erfolgreich) übersetzt
9. Starten des Programms mit Menüauswahl Debuggen→Starten ohne Debuggen (oder bei Bedarf Menüauswahl Debuggen→Debuggen starten)

A.3 Bloodshed Dev-C++

Dev-C++ ist eine freie Entwicklungsumgebung, die um einiges schlanker ist als die umfangreichen MS-Produkte. Sie reicht für die kleinen Projekte, die im Rahmen des Studiums durchgeführt werden, mehr als aus. Die Anleitung wurde mit Version 4.9.9.2 getestet und sollte sich in anderen Versionen nur unwesentlich unterscheiden:

1. Alle Dateien eines Projekts in ein ansonsten leeres Verzeichnis Ihrer Wahl abspeichern
2. Dev-C++ starten
3. Menüauswahl Datei→Neu→Projekt – im sich öffnenden Fenster „Neues Projekt“

   - im Reiter „Basic“ das Icon „Empty Project“ wählen (ggf. Scroll-Balken nach unten ziehen)
   - unten rechts C++-Projekt wählen
   - unten links passenden Namen wählen
   - „Ok“ klicken

   Im sich öffnenden Datei-Dialog

   - in dem im Schritt 1 gewählten Verzeichnis ein neues Verzeichnis erstellen (zweites Icon von oben rechts)
   - in dieses wechseln und
   - durch klicken von „Speichern“ die Projekt-Datei von Dev-C++ in das eben neu erstellte Verzeichnis speichern (in dieses Verzeichnis wird Dev-C++ dann auch Zwischendateien und die fertig kompilierten Programme abspeichern)

4. Häkchen setzen in Menüauswahl Ansicht→Projekt/Klassen-Browser – dort sollte sich im Reiter „Projekt“ das Projekt mit dem in Schritt 3 gewählten Namen befinden
5. Im Kontextmenü (rechte Maustaste) des Projekts den Menüpunkt „Zum Projekt hinzufügen“ wählen – im sich öffnenden Dateidialog

   - das Verzeichnis aus Schritt 1 wählen (in der Regel ist dies eine Ebene über dem aktuellen)
   - dort alle zum Projekt zugehörigen Dateien (Endung `.h` und `.cpp`) auswählen (Mehrfachauswahl bei gehaltener Strg-Taste)
   - „Öffnen“ wählen

6. Im Kontextmenü des Projekts den Menüpunkt „Projekt Optionen“ wählen – im sich öffnenden Fenster

   - Im Reiter „Build Optionen“ als „Ausgabeverzeichnis für Objektdateien“ das in Schritt 3 erstellte Verzeichnis wählen. (Dieser Schritt ist nicht unbedingt notwendig, hat aber den Vorteil, dass alle Dateien, die nicht direkt mit den Quellcode-Dateien (also die mit Endung `.h` oder `.cpp`) zu tun haben, in diesem Extra-Verzeichnis verschwinden.)

7. Menüauswahl Ausführen→Kompilieren + Ausführen (Tastaturkürzel F9) übersetzt und startet das Projekt

Anmerkung: Dev-C++ wartet am Ende nicht automatisch auf einen Tastendruck – man könnte daher manchmal den Eindruck haben, das ausgeführte Programm sei abgestürzt, weil das Anwendungsfenster sich automatisch geschlossen hat. Daher ggf. innerhalb `int main()` vor dem `return 0;` den Befehl `system("Pause");` einfügen, damit das Programm zum Schluss auf einen Tastendruck wartet, bevor das Anwendungsfenster geschlossen wird.

A.4 Kompilieren von der Kommandozeile

Wenn man mit dem Programmieren beginnt, ist es für erste Schritte mitunter hilfreich, auf den vermeintlichen Komfort einer Entwicklungsumgebung zu verzichten und auf Kommandozeilenebene zu arbeiten.

A.4.1 GNU gcc

Unter Linux mit dem frei verfügbaren GNU-C/C++-Compiler lassen sich die Programme leicht übersetzen.

1. Alle Dateien eines Projekts in ein ansonsten leeres Verzeichnis Ihrer Wahl abspeichern
2. Der Aufruf `g++ <datei1.cpp> <datei2.cpp> ... <dateiN.cpp> -o <Programm>` übersetzt das Projekt (die Header-Dateien (Endung `.h`) werden beim Übersetzen nicht explizit mit angegeben)
3. Danach in der Kommandozeile einfach das `<Programm>` aufrufen

A.4.2 MinGW

MinGW ist das „Minimalist GNU for Windows“. Die Anleitung für den GNU-gcc-Compiler funktioniert hier prinzipiell identisch. Die bei der Installation notwendigen Vorarbeiten (und das Übersetzen eines Programms) sind unter `http://www.mingw.org/wiki/MinGWforFirstTimeUsers` beschrieben.

B Begleitmaterialien und Aufgaben für das Labor

B.1 Ein erstes C/C++-Programm

Wir verzichten hier auf das obligatorische „Hello World!“ und erweitern dieses gleich.

```
#include <iostream>
using namespace std;
// diese ersten beiden Zeilen benötigen wir, um Texte ein und ausgeben zu können
// Kommentare beginnen in C++ mit // und gehen bis zum Ende der Zeile

int main(void) { // hier beginnt das eigentliche Programm. Der C/C++-Sprachstandard
  // schreibt "int main" vor, auch wenn manche Compiler hier "void main" akzeptieren,
  // sollten Sie es bei int main belassen

  int zahl; // Variablen (Wertebehälter) müssen dem Compiler vor der Benutzung
  // bekannt gemacht werden (man sagt deklarieren)
  // int gibt an, dass die Variable zahl eine Ganzzahl sein soll

  cout << "Hello! Wie lautet Deine Lieblingszahl?"; // Ausgabe auf der Konsole

  cin >> zahl; // man beachte die Richtung der << und >> je nach Ein- oder Ausgabe

  cout << "Welch ein Zufall, das ist auch meine Lieblingszahl!" << endl;
  // das endl sorgt für einen Zeilenumbruch

  return 0; // vor der abschließenden "}" steht (vorest immer) return 0;
}
```

Wir stellen hier die Ein- und Ausgabe zunächst nach der in C++ üblichen Methode über `iostream`, `cin` und `cout` vor. Die sonst in C übliche Variante (`scanf`, `printf` etc.) benötigt etwas mehr Vorkenntnisse, wir kommen später darauf zurück.

Für alle Programme in den ersten Wochen können Sie die ersten zwei Zeilen, die Zeile mit `int main`, sowie `return 0;` mit der abschließenden `}` unverändert übernehmen. Wir gehen auf deren genaue Bedeutung später ein, übernehmen diese Zeilen zunächst als Rahmen für unsere ersten Programme.

B.2 Die `if`-Anweisung

Fallunterscheidungen sind eine der wesentlichen Kontrollstrukturen in jeder Programmiersprache.

Der Aufbau der `if`-Anweisung: es beginnt mit dem Schlüsselwort `if`, gefolgt von einem von runden Klammern umschlossenen Ausdruck, der sich zu wahr oder falsch auswerten lässt. Es folgen nach einer geschweiften Klammer eine oder mehrere Anweisungen. Nach der schließenden geschweiften Klammer folgt optional das Schlüsselwort `else`, gefolgt von weiteren von geschweiften Klammern umschlossenen Anweisungen[7]. Wir betrachten als Beispiel folgendes Programmfragment:

[7] Folgt nur eine Anweisung, so dürfen die zugehörigen geschweiften Klammern entfallen – dies wird jedoch nicht empfohlen (vgl. Vorlesung)

```
int zahl;
cin >> zahl;
if (zahl==42) {
  cout << "Du hast die Antwort!" << endl;
} else if (zahl==21) {
  cout << "Die halbe Wahrheit." << endl;
} else {
  cout << "Ein schöne Zahl."
}
```

Wir sehen, dass bei Tests auf Gleichheit bei C/C++ zwei `==` verwendet werden (das einfache `=` wird für Zuweisungen benutzt).

Auf Zahlen sind neben den Grundrechenarten `+`, `-`, `*`, `/` (bei der Division wird der Nachkommaanteil einfach gestrichen) insbesondere die Restbestimmung bei ganzzahliger Division definiert – die Restbestimmung wird durch das Operatorsymbol `%` dargestellt. Weitere Operatoren werden wir zu gegebener Zeit kennenlernen.

Übungsaufgaben zur `if`-Anweisung

Geben Sie für jedes C/C++-Programm vorher einen Entwurf als Struktogramm an. Unter `http://www.learn2prog.de` steht ein einfacher Editor für Struktogramme zur Verfügung.

1. Schreiben Sie ein Programm, das eine Jahreszahl einliest und entscheidet, ob das Jahr ein Schaltjahr ist. (Hinweis: ein Jahr ist Schaltjahr, wenn es durch 4, aber nicht durch 100 oder aber durch 400 ganzzahlig teilbar ist.)
2. Schreiben Sie ein Programm, das zu einer Eingabe eines Datums (Tag Monat Jahr) bestimmt, ob das Datum gültig ist oder nicht – beachten Sie hierbei insbesondere, dass in Schaltjahren der 29.2. gültig ist. (Hinweis: Sie können mit `cin` auch mehrere Variablen gleichzeitig einlesen, z. B. liest `cin >> var1 >> var2 >> var3;` drei durch Leerzeichen getrennte Zahlen ein.)

B.3 Allgemeine Hinweise zur Lösung der Übungsaufgaben

Zum Schreiben eines Programmes gehört auch das Testen – schließlich will man (relativ) sicher sein, dass das Programm auch tut, wofür es geschrieben wurde. Überlegen Sie sich also zu jeder Aufgabe, welche Testfälle sinnvoll sind. Am Beispiel der ersten Aufgabe: Sie sollten Ihr Programm für jeden Fall mit mindestens einem Test durchlaufen lassen:

- kein Schaltjahr: z. B. 2010
- „normales“ Schaltjahr: z. B. 2012
- kein Schaltjahr, obwohl durch 4 teilbar: z. B. 2100
- Schaltjahr, obwohl durch 100 teilbar: z. B. 2000

Je nach Programm bieten sich auch noch Randfälle an und sehr kleine oder sehr große Zahlen. Wichtig ist, dass Sie das richtige Ergebnis vorab wissen müssen!

B.4 Einfache Zählschleifen

Wir lernen hier nun auch die erste Art von Schleifen kennen: die Zählschleife. Diese wiederholt eine Anweisung für verschiedene Werte einer Variablen. Hierbei können wir den ersten und letzten Wert angeben und die Schrittweite zwischen zwei aufeinanderfolgenden Wiederholungen. Auch hier ein Beispiel.

```
int zaehle;
for(zaehle=1;zaehle<=10;zaehle=zaehle+3) {
  cout << "Die Zahl " << zaehle << " zum Quadrat lautet " << zaehle*zaehle << endl;
}
```

Als Ausgabe erhalten wir die Quadratzahlen `1`, `16`, `49` und `100`. Üblich ist im obigen Fall die abkürzende Schreibweise `zaehle+=3` für die Angabe der Schrittweite (bei Schrittweite `1` kann man noch kürzer auch `zaehle++` schreiben). Auch hier abstrahieren wir zunächst die Funktionsweise von diesem einen Beispiel. Auf die genaue Bedeutung der Angaben im Kopf der `for`-Schleife gehen wir später ein.

B.5 Unterprogramme

Schon bei kleinen Problemstellungen wächst der Umfang der Programme über eine Bildschirmseite hinaus, sodass man leicht den Überblick verlieren kann. Ebenso tauchen oft dieselben Befehle in derselben Reihenfolge an verschiedenen Stellen im Programm auf. Es wäre also hilfreich, wenn man Programme in kleinere Bausteine zerlegen kann, diesen Bausteinen Namen gibt, die dann als neue C/C++-Befehle verwendet werden können. Neben der bloßen Zusammenfassung von Befehlen zu einem neuen Befehl gibt es die Möglichkeit, das Verhalten durch Parameter zu beeinflussen.

Sollen an verschiedenen Stellen im Programm z. B. die ersten n Quadratzahlen ausgegeben werden, so könnten wir uns eine Funktion

```
void quads(const int n) {
  int zaehle;
  for(zaehle=1;zaehle<=n;zaehle++) {
    cout << "Die Zahl " << zaehle << " zum Quadrat lautet " << zaehle*zaehle << endl;
  }
}
```

schreiben. Solche Funktionen sind im Programmcode vor `int main` einzufügen. Verwendet wird so eine Funktion wie ein normaler C/C++-Befehl: z. B. `quads(12)` zur Ausgabe der ersten 12 Quadratzahlen. Als Parameter kann auch eine Variable oder ein Ausdruck (z. B. `x+4`, wenn `x` eine Variable ist) verwendet werden. Hat eine Funktion mehrere Parameter, so werden diese sowohl bei der Deklaration als auch beim Aufruf durch Kommata voneinander getrennt (die Anzahl der Parameter in der Deklaration und beim Aufruf muss übereinstimmen, die Parameter beim Aufruf werden von links nach rechts den Parametern bei der Deklaration zugeordnet); hat die Funktion keinen Parameter, so verbleiben lediglich die runden Klammern `()` (um auch optisch zu verdeutlichen, dass die Funktion keine Parameter hat, sollte man `(void)` schreiben).

Soll die Funktion nicht nur eine Ausgabe liefern, sondern ein Ergebnis berechnen und in einer Variablen ablegen, so kann man die Methoden des strukturierten Entwurfs aus der Vorlesung

verwenden. Diese haben den großen Vorteil, dass sie für beliebige Anzahlen an zu berechnenden Werten einheitlich anzuwenden sind und nebenbei auch modellierte Lese- und Schreibberechtigungen aus dem Entwurf automatisch mit umsetzen. C/C++ besitzt noch ein weiteres Rückgabekonzept; wir führen dies hier auch nur an einem Beispiel ein. Folgende Funktion berechnet die Summe der natürlichen Zahlen von 1 bis n:

```
int gauss(const int n) {
  int ergebnis=0, zaehle;
  for(zaehle=1;zaehle<=n;zaehle++) {
    ergebnis+=zaehle;
  }
  return ergebnis; // wir hätten auch gleich return n*(n+1)/2; schreiben können :-)
}
```

Das `int` vor dem Namen der Funktion gibt an, welchen Datentyp das Ergebnis der Funktion haben wird, das Ergebnis wird über den Befehl `return` an das aufrufende Programm übermittelt (das `return` kann auch an mehreren Stellen in der Funktion stehen, z. B. in den verschiedenen Zweigen einer Fallunterscheidung – wichtig ist lediglich, dass in jedem Ablauf der Funktion ein `return` erreicht werden kann – beachten Sie, dass nach einem `return` keine Befehle dieser Funktion mehr ausgeführt werden.

Der Aufruf berechnet einen Wert, dieser muss entweder einer Variablen zugewiesen werden oder in einer Anweisung verwendet werden, sonst ist der berechnete Wert verloren. Der Aufruf `gauss(6);` würde also nichts nachhaltig bewirken (die Rechenschritte werden aber trotzdem ausgeführt; beinhaltet die Funktion eine Ausgabe auf der Konsole, so wird diese ausgeführt), der Aufruf `g=gauss(6);` speichert in der (vorher zu deklarierenden) Variablen `g` den Wert `21` ab, der Aufruf `cout << gauss(6);` gibt den Wert `21` direkt auf der Konsole aus.

Offensichtlich kann über diesen `return`-Mechanismus nur ein Wert zurückgegeben werden, dies ist der entscheidende Nachteil gegenüber der in der Vorlesung vorgestellten Methode.

Funktionen, die mit `return` arbeiten, haben lediglich einen Vorteil, wenn das Ergebnis nur genau einmal verwendet werden soll, z. B. bei einer Ausgabe oder in der Bedingung einer `if`-Abfrage.

Übungsaufgaben zur `for`-Schleife und Unterprogrammen

Geben Sie für jedes C/C++-Programm vorher einen Entwurf als Struktogramm an. Unter `http://www.learn2prog.de` steht ein einfacher Editor für Struktogramme zur Verfügung.

1. Schreiben Sie ein Programm, das ein Datum einliest und berechnet, am wievielten Tag im Jahr das Datum ist.

2. Schreiben Sie ein Programm, das ein Anfangs- und ein Enddatum einliest. Ist das Enddatum später als das Anfangsdatum, so soll die Anzahl der Tage berechnet werden, sonst eine Fehlermeldung ausgegeben werden. Überlegen Sie sich, welche Teilprobleme hier zu lösen sind und fassen Sie diese zu geeigneten Unterprogrammen zusammen.

3. Zusatzaufgabe: Schreiben Sie ein Programm, das zu einem Datum angibt, auf welchen Wochentag dieses Datum fällt. (Hinweis: der fast weltweit gültige Gregorianische Kalender begann am Freitag, den 15. Oktober 1582.)

4. Zusatzaufgabe: Untersuchen Sie die Wochentage vom 1.1.1600, 1.1.2000 und 1.1.2400 – was fällt auf? Untersuchen Sie wie häufig in einem 400-Jahre-Zyklus der 13. eines jeden Monats auf den Montag, Dienstag, ... oder Sonntag fällt.

5. Zusatzaufgabe: Überlegen Sie sich, welche Art von Konstrukten Sie sich gewünscht hätten, um eine oder mehrere der obigen Aufgaben einfacher lösen zu können. Welche Ideen hatten Sie, die Sie mangels Kenntnis der betreffenden Konstrukte nicht haben umsetzen können?

B.6 Kopf- und fußgesteuerte Schleifen – und – Zahlendarstellung im Rechner

Bei der Ein- und Ausgabe von Zahlen im Rechner arbeiten wir mit der uns geläufigen Dezimaldarstellung. Rechnerintern sind diese Zahlen aber binär dargestellt, nur mit 0en und 1en – glücklicherweise ist dies für den Programmierer und Benutzer aber transparent, wir müssen uns beim Programmieren darüber keine Gedanken machen, wollen uns die Darstellung aber einmal am Beispiel einer vorzeichenlosen Ganzzahl (`unsigned int`) bewusst machen. Die 37 im Dezimalsystem wird binär 100101 dargestellt – $100101 = 1{\cdot}2^5 + 0{\cdot}2^4 + 0{\cdot}2^3 + 1{\cdot}2^2 + 0{\cdot}2^1 + 1{\cdot}2^0$. Gerechnet wird stellenweise wie im Dezimalsystem, z. B. bei der Addition:

```
          100101  (=37)
        +  10111  (=23)
Übertrag   111
        = 111100  (=60)
```

Viele Hinweise zu Binärzahlen finden Sie unter `http://de.wikipedia.org/wiki/Dualsystem`

Zählschleifen verwendet man immer, wenn die Anzahl der Schleifendurchläufe vorab bekannt ist. Die zugehörige `for`-Schleife haben wir bereits kennengelernt. In allen anderen Fällen wiederholt man einen Anweisungsblock solange (kopfgesteuert) oder bis (fußgesteuert) eine Bedingung erfüllt ist.

Betrachten wir hier nun folgende Funktion:

```
void dezToDual(const unsigned int z) {
  unsigned int zahl=z;
  while (zahl>0) {
    if ((zahl%2)==0) {
      cout << "0";
    } else {
      cout << "1";
    }
    zahl=zahl/2;
  }
}
```

Wird diese Funktion mit Parameter `37` aufgerufen, so erhalten wir `101001` – die umgedrehte Binärdarstellung des Parameters. Dies ist kein Zufall – überlegen Sie selbst, warum. Für die `0` erhalten wir keine Ausgabe – eigentlich hätten wir eine fußgesteuerte Schleife gebraucht, sodass der Schleifenrumpf mindestens einmal ausgeführt wird. Hierzu können wir das Konstrukt `do { <Anweisung> } while (<Bedingung>);` benutzen. Schreiben Sie obige Funktion entsprechend um.

Übungsaufgaben zur `while`-Schleife

Geben Sie für jedes C/C++-Programm vorher einen Entwurf als Struktogramm an. Unter `http://www.learn2prog.de` steht ein einfacher Editor für Struktogramme zur Verfügung.

1. Als Kind spielt man manchmal Zahlenraten: Ein Kind denkt sich eine natürliche Zahl zwischen 0 und 100 aus, das andere Kind versucht die Zahl zu erraten und erhält als Hilfestellung lediglich den Hinweis, ob die gesuchte Zahl kleiner oder größer ist.

   Schreiben Sie ein Programm, das eine von Ihnen ausgedachte Zahl mit möglichst wenig Versuchen errät. Nach jedem Vorschlag des Programms wird der Benutzer gefragt, ob die Zahl erraten wurde. Wenn der Benutzer versucht, das Programm hereinzulegen (durch widersprüchliche Angaben), so soll das Programm dies erkennen. Wieviel Versuche benötigt Ihr Programm maximal bis die Zahl erraten wurde?

2. Zusatzaufgabe: Wie würden Sie das Programm vorgehen lassen, wenn der Bereich, aus dem die Zahl stammt, vorab nicht bekannt ist? Modifizieren Sie Ihre Ratefunktion entsprechend.

3. Berechnung der Wurzel einer Zahl z: Das Verfahren, das wir beim Zahlenraten anwenden, nennt sich binäre Suche (oder Intervallschachtelung). Wir können dieses Prinzip auch benutzen, um die Wurzel w einer Zahl zu bestimmen – ist $w * w > z$, so ist die geratene Zahl w zu groß, bei $w * w < z$ zu klein. Schreiben Sie eine Funktion, die mit Hilfe der Intervallschachtelung die Wurzel einer Zahl bestimmt. Hinweis: da sich die Wurzel (außer bei Quadratzahlen) nicht exakt im Rechner darstellen lässt, sind wir auf Näherungen angewiesen – brechen Sie Ihre Berechnung geeignet ab und verwenden Sie einen möglichst genauen Gleitkommazahlentyp (z. B. `double`).

4. Ein weiteres Verfahren zur Bestimmung der Wurzel benutzt eine ähnliche Formel: Ausgehend von $w_0 = z$ berechnen wir eine Folge von Zahlen $w_{i+1} = \frac{1}{2}(w_i + \frac{z}{w_i})$. Diese Zahlenfolge nähert sich (von oben) der Wurzel an. Dieses Verfahren heißt Heron-Verfahren. Schreiben Sie ein Programm, das die Wurzel nach dem Heron-Verfahren annähert.

5. Schreiben Sie ein Programm, dass die beiden Verfahren zur Wurzelberechnung vergleicht, indem Sie jeweils angeben, wie groß die Abweichung des Quadrats der geratenen Wurzel w zur Zahl z ist ($z-w*w$). Welches Verfahren würden Sie nach dieser Analyse verwenden?

B.7 Eigene Datentypen mit `typedef`, `struct` und `enum`

Mit `typedef` kann man gültigen Datentypen einen neuen, zusätzlichen Namen geben, z. B. `typedef float Temperatur;`. Dies dient neben der Lesbarkeit von Programmen der leichteren Wartbarkeit, falls der Datentyp in Teilen eines Programms verändert werden soll. Man kann mit `typedef` auch eigene Arraytypen definieren: `typedef int dreiints[3];` definiert einen Datentyp für ein 3-elementiges Array[8] – die über `dreiints tupel;` deklarierte Variable `tupel` ist dann ein Array. Als Nebeneffekt haben so definierte Arraytypen den Vorteil, dass die Parameterübergabe dann bei Arrays syntaktisch identisch mit der einfacher Datentypen ist (beschränkt man sich auf die Methode des strukturierten Entwurfs, d. h., Variablen mit ausschließlichem Lesezugriff werden mit `const <typ> <name>` übergeben und Variablen

[8] Leider ist die Syntax von `typedef` hier verwirrend: die Anzahl der Elemente im Array wird an den Namen des neuen Datentyps angehängt.

mit Schreibzugriff mit `<typ> & <name>`, so ist auch die Semantik identisch wie in der Vorlesung beschrieben). Verbunde (Schlüsselwort `struct`) bieten die Möglichkeit beliebige Variablen zu einer Datenstrukur zusammenzufassen. Bei der Programmierung ist es üblich, dass man `struct`-Variablen auch bei ausschließlichem Lesezugriff stets als Referenz übergibt, da sonst die meist umfangreichen Daten zunächst kopiert werden müssten. Ist bei der Variablen nur Lesezugriff vorgesehen, so kann dies bei der Parameterübergabe durch ein zusätzliches `const` kenntlich gemacht werden (`const <struct_typ> & <name>`) – so spart man das Erstellen der Kopie, ohne die Zugriffsrechte aus dem Datenflussdiagramm zu übergehen.

Schließlich bieten Aufzählungstypen eine weitere Möglichkeit, eigene Datentypen zu definieren. Die Zeile `enum Wochentag {Mon,Die,Mit,Don,Fre,Sam,Son};` definiert z. B. einen eigenen Datentyp `Wochentag`. Im Wesentlichen entspricht dies `typedef int Wochentag;` und den Präprozessoranweisungen `#define Mon 0`, `#define Die 1`, `#define Mit 2`, `#define Don 3`, `#define Fre 4`, `#define Sam 5` und `#define Son 6` – es werden also lediglich Konstanten definiert, die mit einem eindeutigen Namen angesprochen werden können. Bei reinem C lassen sich einer Variablen vom Typ `Wochentag` dann auch andere Werte als 0 bis 6 zuweisen, C++ lässt standardmäßig nur die Zuweisung der vordefinierten Werte zu. Auch hier dienen diese Aufzählungstypen der Les- und Wartbarkeit. Während Wochentage naturgemäß auf sieben beschränkt sind, kann man sich leicht andere Anwendungen vorstellen, z. B. eine Auflistung von vordefinierten Farben, bei denen dann zusätzliche Farben ohne Aufwand hinzugefügt werden können (ohne Aufzählungstypen würden wir die Farbnamen anderweitig kodieren, was bei Erweiterungen schnell zu größerem Änderungsbedarf führen kann).

Alle selbstdefinierten Datentypen sollten hinter den `#include`-Anweisungen und vor der Definition eigener Funktionen eingefügt werden.

Übungsaufgaben zu `typedef`, `struct` und `enum`

Geben Sie für jede C/C++-Funktion vorher einen Entwurf als Programmablaufplan mit zugehörigem Datenflussdiagramm (Schnittstelle) an.

Wir greifen hier nochmal die Berechnung des Wochentages zu einem Datum auf. Zur Erinnerung: der Gregorianische Kalender begann am 15.10.1582 – dies war ein Freitag.

In diesem Programm lassen sich folgende Teilprobleme identifizieren:

- Eingabe eines Datums,
- Überprüfen der Gültigkeit eines Datums,
- die eigentliche Berechnung des Wochentages,
- die Ausgabe des Datums und
- die Ausgabe des Wochentages.

Benutzen Sie in Ihrem Programm folgendes Fragment (die Signatur der Funktion `istGueltig` lautet `bool istGueltig(const Datum dat)`):

```
int main(void) {
  Datum dat;
  Wochentag wotag;
  do { // wiederhole, bis gültiges Datum eingegeben wird
    Datumseingabe(dat);
  } while (!(istGueltig(dat)));
  berechneWochentag(dat,wotag);
  cout << "Am ";
  AusgabeDatum(dat);
  cout << " war ein ";
  AusgabeWochentag(wotag);
  cout << "." << endl;

  return 0;
}
```

Aufgabe: Definieren Sie sich einen eigenen Datentyp `Datum` (der Datentyp `Wochentag` kann von oben übernommen werden). Geben Sie neben den Programmablaufplänen und Implementierungen der Teilprobleme auch ein Programmhierarchiediagramm an.

B.8 Programmhierarchie, Erstellung und Test von umfangreichen Programmen

Fast alle Probleme, die wir mit Hilfe von Computern lösen, sind zu umfangreich, als dass ein Programm in wenigen Zeilen dafür geschrieben wäre. So beginnt ein Entwurfsprozess in der Regel damit, das Problem in sinnvolle und möglichst unabhängig voneinander zu lösende Teilprobleme zu gliedern. Sind die Teilprobleme benannt, können die Aufrufabhängigkeiten in einem Programmhierarchiediagramm festgehalten und die zugehörigen Schnittstellen festgelegt werden. Sind die Datentypen entworfen, so können danach die Teilprobleme unabhängig voneinander gelöst werden. Insbesondere sollte man stets auch jedes Teilproblem unabhängig von den anderen zunächst für sich testen. Erst, wenn man sicher ist, dass die Teilprobleme in sich korrekt gelöst sind, ist ein Test des Gesamtsystems erfolgversprechend. Alles andere ist Stochern im Nebel.

Man unterscheidet zunächst statische und dynamische Tests. Statische Testmethoden setzen direkt auf dem Source-Code auf und überprüfen z. B. die Syntax (dies übernimmt der Compiler). Weitere statische Testmethoden sind Checklisten (Überprüfung des Codes auf lokale Plausibilität: passen die Operanden zu den Operatoren (C/C++ erkennt nicht alles), sind Präferenzen der Operatoren berücksichtigt, korrekte Klammerung beachtet, ist Division durch Null möglich, etc.) und Codereview (Überprüfung, ob die Lösungsmethode der Problemstellung angemessen ist, ob die Umsetzung sinnvoll ist, etc.). Es gibt auch formale Verifikationsmöglichkeiten (Stichwort „Hoare-Kalkül“), die ein Programm als eine Funktion auffassen und versuchen Beziehungen zwischen Variablen durch mathematische Formeln zu beschreiben.

Zum dynamischen Testen von Teilproblemen und auch des Gesamtprogramms gehören eine ausreichende Anzahl von Testfällen. Für diese muss dem Tester die korrekte Antwort vorliegen (man muss das Teilproblem also auch mal von Hand lösen oder Eingaben testen, für die die korrekte Lösung bereits bekannt ist). Dynamische Testmethoden werden grob in Black-Box-Tests und White-Box-Tests unterschieden. Beim Black-Box-Test werden Testfälle ohne Kenntnis der Implementierung entworfen. Sinnvoll ist dabei eine Auswahl aus normalen Werten, Extremwerten (Eingaben für Randfälle oder Eingaben, für die das Ergebnis ein Randfall wird) und – wenn das Programm auf Robustheit überprüft werden soll – auch fehlerhafte Eingaben. Für jeden Fall muss das erwartete Ergebnis vorher dokumentiert sein.

Der White-Box-Test geht einen Schritt weiter und wählt die Testfälle mit Kenntnis des Quellcodes aus: Hierbei unterscheidet man abhängig von der Abdeckung der möglichen Programmabläufe verschiedene Stufen – Zeilenüberdeckung: Ausführung jeder Quellcodezeile; Anweisungsüberdeckung: Ausführung jeder Anweisung; Zweig- oder Kantenüberdeckung: Durchlaufen aller möglichen Kanten im zugehörigen Programmablaufplan; Bedingungs- oder Termüberdeckung: Durchlaufen aller relevanten Variablenbelegungen in Bedingungen; Pfadüberdeckung: Betrachtung aller Pfade durch ein Modul. Sie sollten hier zumindest die Zweigüberdeckung anstreben.

B.9 Umfangreiche Übungsaufgabe: Lösung von Umfüllproblemen

Früher oder später stößt jede(r) mal auf Knobelaufgaben, bei denen meist drei unterschiedlich große Gefäße vorliegen, von denen eines gefüllt ist, und man eine bestimmte Menge auf die Einzelgefäße verteilen soll. Der Klassiker: die drei Gefäße haben eine Kapazität von 8, 5 und 3 Liter, das 8-Liter-Gefäß ist gefüllt und man soll davon 4 Liter in das 5-Liter-Gefäß füllen. Da keine Markierungen an den Gefäßen sind, kann beim Umfüllen von einem in ein anderes Gefäß dieses nur komplett umgefüllt werden oder das letztere Gefäß jeweils bis zum Rand gefüllt werden. Um in Zukunft nicht mehr mit solchen Problemen belästigt zu werden, schreiben wir uns ein Programm zur Lösung: Eingabe: die Größe der drei Gefäße, der aktuelle Inhalt zu Beginn und der gewünschte Inhalt am Ende. Das Programm soll die Lösung finden und ausgeben.

Wir geben mehrere Teilaufgaben an, die Sie nacheinander zu der Lösung des Gesamtproblems zusammensetzen können. Üben Sie an diesem Beispiel auch Programmhierarchiediagramme und geben Sie für jedes Teilproblem ein Datenflussdiagramm und einen Programmablaufplan an.

Verwaltung einer Menge: Wir werden erreichbare Mengenverteilungen in den Gefäßen verwalten müssen. Zu Beginn sind 8 Liter im ersten Gefäß, die anderen beiden sind leer – dies lässt sich als Tupel `(8,0,0)` beschreiben (wir nennen solch ein Tupel eine Konfiguration). Im ersten Umfüllschritt können wir entweder `(3,5,0)`, `(5,0,3)` oder `(0,0,0)` erreichen – jenachdem, ob wir das erste Gefäß in das zweite oder dritte gießen oder es ausleeren. Bevor wir uns um den Datentyp zur Verwaltung der Gefäßinhalte kümmern, betrachten wir zunächst die Mengenverwaltung an sich (hier kann man sich z. B. mit `typedef int Elementtyp;` einen einfachen Typ erstellen, um zunächst die Mengenverwaltung zu schreiben und zu testen).

In der Vorlesung wurde eine einfache Datenstruktur für eine Menge angegeben (ein Verbund, bestehend aus einem ausreichend großen Array und einer Zahl, die angibt, wie viele Elemente die Menge enthält).

```
typedef ... Elementtyp; // setzen Sie für ... den zu verwaltenen Typ ein
const int N_MAX=1000; // maximale Anzahl der Elemente in der Menge
struct Menge {
  Elementtyp inhalt[N_MAX];
  int anzElems;
};
```

Wir müssen selbst dafür sorgen, dass gleiche Elemente nicht mehrfach in der Menge vorkommen. Schreiben Sie für solch einen Datentyp `Menge` vier Funktionen:

- Initialisierung einer leeren Menge (verwirft die möglicherweise vorhandenen Elemente)
- ein Test, ob ein Element in der Menge bereits vorhanden ist
- das Hinzufügen eines Elementes, das noch nicht in der Menge ist
- die Ausgabe der Elemente in der Menge
- Zusatzaufgabe: das Löschen eines Elementes aus der Menge (diese Funktion ist für eine Mengenverwaltung nützlich, wir werden wir sie für die Lösung der Umfüllprobleme jedoch nicht benötigen)

Sie brauchen sich hier nicht um die Reihenfolge der Elemente zu kümmern. Es ist insbesondere keine Sortierung der Elemente erforderlich. Testen Sie Ihre Funktionen zunächst mit einem einfachen Datentyp, z. B. `int`.

Testen Sie nun auch mit komplizierteren Datentypen, z. B. das oben genannte `dreiints`. Schreiben Sie dazu eine Schleife, die die Werte `(0,0,0)`, `(1,1,1)`, ..., `(9,9,9)` in die Menge einfügt. Hinterfragen Sie Ihre Beobachtungen und überlegen Sie, wie dies mit dem in der Vorlesung Gelernten zusammenhängt. Testen Sie abschließend mit einem auf `struct` basierten Datentyp (z. B. `Datum`, der aus drei (einzelnen) `int`-Variablen besteht).

Umfüllen von einem Gefäß in ein anderes: Hierfür sind die Kapazität des Zielgefäßes und die Inhalte des Start- und Zielgefäßes ausschlaggebend. Schreiben Sie eine Funktion, die auf Basis dieser drei Werte angibt, welche Inhalte die beiden Gefäße danach haben. Beachten Sie die Fallunterscheidung, dass das Startgefäß entweder vollständig in das Zielgefäß gefüllt werden kann oder dass aufgrund der Kapazität des Zielgefäßes dieses voll gefüllt wird und ein Teil im Startgefäß verbleibt (dieser Fall trat im obigen Beispiel bei `(8,0,0)`→`(3,5,0)` und `(8,0,0)`→`(5,0,3)` auf).

Mögliche Folgekonfigurationen bestimmen: Hier soll nun ausgehend von einer Konfiguration die Menge der Folgekonfigurationen ermittelt werden. Dies sind bis zu neun verschiedene: jedes Gefäß kann in eines der beiden anderen umgefüllt oder (in den Ausguss) geleert werden. Die Folgekonfigurationen zu `(6,3,2)` sind mit obigen Kapazitäten: `(4,5,2)`, `(5,3,3)`, `(0,3,2)`, `(8,1,2)`, `(6,2,3)`, `(6,0,2)`, `(8,3,0)`, `(6,5,0)` und `(6,3,0)`. Es reicht hier die Ausgabe der erzeugten Elemente auf der Konsole zum Testen der Korrektheit.

Prüfen, ob es eine Lösung gibt: Ausgehend von einer Startkonfiguration können wir nun Folgekonfigurationen erzeugen und diese in der Menge speichern. Durch den Vergleichstest werden keine Konfigurationen doppelt aufgenommen. Solange noch nicht betrachtete Konfigurationen in der Menge sind, berechne deren Folgekonfigurationen und trage diese in die Menge ein. Wird die Zielkonfiguration erreicht, so gibt es eine Lösung. Schon betrachtete Konfigurationen bleiben in der Menge, sodass diese später nicht nochmals aufgenommen werden. Beispiel: zu den Kapazitäten `(6,4,3)` soll mit der Startkonfiguration `(6,0,0)` die Zielkonfiguration `(3,3,0)` erreicht werden. Es wird die Menge mit `(6,0,0)` initialisiert und die Folgekonfigurationen `(2,4,0)`, `(3,0,3)` und `(0,0,0)` werden hinzugefügt (beim „Umfüllen“ der Gefäße 2 und 3 entsteht hier nur die schon bekannte Startkonfiguration `(6,0,0)`). Für `(2,4,0)` werden die Konfigurationen `(0,4,2)`, `(0,4,0)`, `(2,1,3)` und `(2,0,0)` hinzugefügt, für `(3,0,3)` die Konfigurationen `(0,3,3)`, `(3,3,0)` (und ggf. noch `(3,0,0)`), sodass nun die Zielkonfiguration `(3,3,0)` erreicht ist.

Ausgabe des Lösungsweges: Um einen Lösungsweg angeben zu können, müssen wir uns merken von welcher Konfiguration kommend wir eine Folgekonfiguration zu der Menge hinzugefügt haben. Im obigen Beispiel hätten wir zu `(3,3,0)` den Vorgänger `(3,0,3)` vermerken können, dieser hätte als Vorgänger die Startkonfiguration `(6,0,0)` gehabt. Erweitern Sie den in der Menge verwalteten Datentyp, sodass Sie sich den Vorgänger merken können. Beachten Sie, dass gleiche Konfigurationen mit einem anderen Vorgänger nicht zusätzlich in die Menge aufgenommen werden müssen. Die Lösung lässt sich dann leicht von der Zielkonfiguration aus rückwärts bis zur Startkonfiguration ausgeben.

Die Lösung für das Umfüllen hat die zusätzliche Eigenschaft, dass diese eine minimal mögliche Anzahl von Schritten benötigt. Überlegen Sie, warum dies so ist.

Ideen zur Erweiterung: Das Problem lässt sich beliebig erweitern. Hier einige Vorschläge, selbst wenn man dabei bei drei Gefäßen bleibt:

- Manchmal sind nur die Inhalte zweier Gefäße vorgegeben (z. B. Aufteilen der 8 Liter auf 2 mal 3 Liter), wobei zum Teil nicht festgelegt ist, dass sich diese in den ersten beiden Gefäßen befinden müssen, oder in manchen Gefäßen ist der Reststand egal.
- In manchen Problemstellungen hat eines der drei Gefäße keine Kapazität (Flüssigkeit kommt aus der Leitung oder wird in den Ausguss gekippt).
- In manchen Problemstellungen hat eines der drei Gefäße eine „hohe“, aber nicht näher spezifizierte Kapazität – meist ist dieses Gefäß zu Beginn gefüllt. Diese Problemstellung ist ähnlich, aber nicht identisch zur ersten Variation. Lösen Sie hiermit folgende Aufgabe: Neben einem großem Fass mit Wasser stehen zwei zu Beginn leere Gefäße mit 5 und 3 Litern zur Verfügung. Füllen Sie in das 5- und 3-Liter-Gefäß jeweils 1 Liter ab (gleichzeitig, nicht nacheinander).
- Finden Sie Kapazitäten und Startkonfigurationen, sodass die gefundene (kürzeste) Lösung möglichst viele Schritte braucht. Beschränken Sie sich dabei zunächst auf eine maximale Kapazität von 9 pro Gefäß. Mit welcher Laufzeit müssen Sie hier rechnen?

Erweitern Sie Ihr Programm geeignet für solche Problemstellungen oder entwickeln Sie eigene Ideen.

B.10 Modularisierung

Auch bei der Modularisierung gibt es ein Standardvorgehen, das nach einem Schema arbeitet und immer funktioniert. Ausgangspunkt ist dabei das Datenhierarchiediagramm (DH). Für jede Datenstruktur (und sei sie noch so unbedeutend) wird ein Dateipaar `.h` / `.cpp` erstellt. Man beginnt wie bei der Umsetzung des DH üblicherweise von unten. Jede Headerdatei ist dabei vom Aufbau identisch – zur Datenstruktur `beispiel` wird eine Datei `beispiel.h` erstellt mit Dateiinhalt

```
#ifndef BEISPIEL_H
#define BEISPIEL_H

// ist die Datenstruktur Beispiel ein typedef und
// ist die dabei verwendete Struktur kein einfacher Datentyp
#include "verwendeteDatenstruktur.h"

// sonst ist die Datenstruktur ein struct und
// für jeden Selektor, der nicht auf einem einfachen Datentyp aufbaut
#include "verwendeteDatenstruktur1.h"
#include "verwendeteDatenstruktur2.h"
...
#include "verwendeteDatenstrukturN.h"

// falls noch Datentypen aus anderen Bibliotheken verwendet werden
// diese ebenfalls hier einbinden, z.B.
#include <iostream>
// dies ist hier nicht nötig,
// wenn z.B. <iostream> nur innerhalb der Funktionen verwendet wird

// es folgen für jede Funktion, die die Datenstruktur beispiel und
// ausschließlich die oben eingebundenen Datenstrukturen in der Signatur benutzen
// die Signaturen der Funktion
// üblicherweise verzichtet man auf die Parameternamen,
// bei vielen gleichartigen Parametern empfiehlt sich, über eine
// Kommentarzeile die Bedeutung der Parameter zu verdeutlichen
void fkt1(...); // Rumpf der Funktion entfällt hier, stattdessen ein ';'
void fkt2(...);
...
void fktM(...);

// abschließend kommt ein
#endif
// dieses schließt die Klammer zum obigen #ifndef
```

In der Datei mit Endung `.cpp` steht in der ersten Zeile `#include "beispiel.h"`. Wenn die zugehörigen Funktionen noch weitere Bibliotheken verwenden, die in `beispiel.h` nicht eingebunden wurden, so müssen diese hier noch eingebunden werden (z. B. `iostream`). Schließlich folgt noch die Implementierung der Funktionen.

Manchmal enthält ein Modul Hilfsfunktionen, die nur innerhalb des Moduls aufgerufen werden, aber von anderen Modulen nicht einzeln aufgerufen werden. In diesem Fall empfiehlt es sich, die betreffenden Funktionen nur in der `.cpp`-Datei zu implementieren und in `.h` nicht zu erwähnen.

Desweiteren gibt es manchmal Hilfsfunktionen, die auf nicht selbstdefinierten Datentypen arbeiten. Sind dieses Hilfsfunktionen, die nur im Kontext eines selbstdefinierten Datentyps sinnvoll sind, werden diese dem entsprechenden Dateipaar `.h` / `.cpp` zugeordnet (ggf. nur in der `.cpp`-Datei), anderenfalls werden diese Funktionen – thematisch gruppiert – in geeignete Module aufgeteilt.

Wir betrachten das Vorgehen an folgendem Beispiel eines Programms, das mit Doppelbrüchen arbeitet: Es wird ein `struct Bruch` und ein `struct Doppelbruch` (mathematisch betrachtet ein großer Bruchstrich mit einem Bruch oberhalb und einem Bruch unterhalb) eingeführt. Zum Kürzen von Brüchen benötigen wir den größten gemeinsamen Teiler von Zähler und Nenner eines Bruches – hier der übliche Algorithmus von Euklid (wir lassen uns von der rekursiven Formulierung nicht irritieren). Zusätzlich wird als Funktionalität ein Vereinfachen eines Doppelbruchs in einen einfachen Bruch und das Addieren zweier Brüche zur Verfügung gestellt, sowie `cout` aus der `iostream`-Bibliothek für die Ausgabe von Brüchen und Doppelbrüchen erweitert.

```
#include<iostream>
using namespace std;

struct Bruch {
  int z,n;
};

struct Doppelbruch {
  Bruch oben,unten;
};

int ggT(const int a, const int b) {
  if (b==0) { return a; }
  else { return ggT(b,a%b); }
}

void kuerze(Bruch &a) {
  int kurz=ggT(a.n,a.z);
  a.n/=kurz;
  a.z/=kurz;
}

Bruch vereinfache(Doppelbruch &a) {
  Bruch erg;
  erg.z=a.oben.z*a.unten.n;
  erg.n=a.oben.n*a.unten.z;
  kuerze(erg);
  return erg;
}
```

```
Bruch operator+(const Bruch&a, const Bruch&b) {
  Bruch c;
  c.n=a.n*b.n; c.z=a.z*b.n+b.z*a.n;
  kuerze(c);
  return c;
}

ostream& operator << (ostream& os, const Bruch&a) {
  os << '(' << a.z << '/' << a.n << ')';
  return os;
}

ostream& operator << (ostream& os, const Doppelbruch&a) {
  os << '(' << a.oben << '/' << a.unten << ')';
  return os;
}

int main(void) {
  Bruch a={2,3}, b={1,12};
  cout << a << '+' << b << '=' << a+b << endl;

  Doppelbruch c={{2,3},{1,12}};
  cout << c << '=' << vereinfache(c) << endl;

  return 0;
}
```

Wir identifizieren hier zunächst die Funktion `ggT(.,.)`, die nur auf Standarddatentypen arbeitet. Da wir diese sicher auch in anderen Programmen wiederverwenden wollen, sehen wir für diese ein eigenes Modul für mathematische Hilfsfunktionen vor:

Datei `mathilfun.h`**:**

```
#ifndef MATHILFUN_H
#define MATHILFUN_H

int ggT(const int, const int);

#endif
```

Datei `mathilfun.cpp`**:**

```
#include "mathilfun.h"

int ggT(const int a, const int b) {
  if (b==0) { return a; }
  else { return ggT(b,a%b); }
}
```

Das Standardvorgehen sieht nun für jedes `struct` ein eigenes Dateipaar `.h` / `.cpp` vor. Die Zuordnung der Funktionen ist meist eindeutig. In obigem Beispiel bleibt lediglich für das Vereinfachen eines Doppelbruchs zu einem (einfachen) Bruch die Wahl: sinnvollerweise nimmt man hierbei das Modul mit der „komplizierteren“ Datenstruktur (d. h. in diesem Beispiel auch, dass das Modul für den Doppelbruch das Modul für Bruch einbinden muss).

Datei `bruch.h`**:**

```
#ifndef BRUCH_H
#define BRUCH_H

#include<iostream>
using namespace std;

struct Bruch {
  int z,n;
};

void kuerze(Bruch &);
Bruch operator+(const Bruch&, const Bruch&);
ostream& operator << (ostream&, const Bruch&);

#endif
```

Datei `bruch.cpp`**:**

```
#include "bruch.h"
#include "mathilfun.h"

void kuerze(Bruch &a) {
  int kurz=ggT(a.n,a.z);
  a.n/=kurz;
  a.z/=kurz;
}

Bruch operator+(const Bruch&a, const Bruch&b) {
  Bruch c;
  c.n=a.n*b.n; c.z=a.z*b.n+b.z*a.n;
  kuerze(c);
  return c;
}

ostream& operator << (ostream& os, const Bruch&a) {
  os << '(' << a.z << '/' << a.n << ')';
  return os;
}
```

Datei `doppelbruch.h`**:**

```
#ifndef DOPPELBRUCH_H
#define DOPPELBRUCH_H

#include "bruch.h"
#include<iostream>
using namespace std;

struct Doppelbruch {
  Bruch oben,unten;
};

Bruch vereinfache(const Doppelbruch &);
ostream& operator << (ostream&, const Doppelbruch&);

#endif
```

Datei `doppelbruch.cpp`**:**

```
#include "doppelbruch.h"

Bruch vereinfache(const Doppelbruch &a) {
  Bruch erg;
  erg.z=a.oben.z*a.unten.n;
  erg.n=a.oben.n*a.unten.z;
  kuerze(erg);
  return erg;
}

ostream& operator << (ostream& os, const Doppelbruch&a) {
  os << '(' << a.oben << '/' << a.unten << ')';
  return os;
}
```

„`int main`“: Unsere Datei mit dem Hauptprogramm ist nun sehr übersichtlich geworden:

```
#include"bruch.h"
#include"doppelbruch.h"
#include<iostream>
using namespace std;

int main(void) {
  Bruch a={2,3}, b={1,12};
  cout << a << '+' << b << '=' << a+b << endl;

  Doppelbruch c={a,b};
  cout << c << '=' << vereinfache(c) << endl;

  return 0;
}
```

Aufgabe zur Modularisierung: Teilen Sie die Funktionen Ihres Programms zur Lösung von Umfüllproblemen in mehrere Module auf:

- ein Modul `menge` soll dabei die Mengenverwaltung beinhalten
- ein Modul `umfuellen` die Funktionalität zur Lösung von Umfüllproblemen (dieses Modul wird das Modul zur Mengenverwaltung mit Hilfe von `#include "menge.h"` einbinden)
- beim Standardvorgehen (für jede Datenstruktur des Datenhierarchiediagramms ein Dateipaar `.h` / `.cpp`) werden noch mehr Module entstehen – überlegen Sie, inwieweit sich dieses in Ihrer konkreten Umsetzung vermeiden lässt
- das Hauptprogramm, das die Kapazitäten, den Inhalt der Startgefäße und das Ziel einliest, sowie damit die Lösungsroutine aus dem Modul `umfuellen` aufruft

Beachten Sie, dass die Aufteilung so vorgenommen wird, dass die Mengenverwaltung leicht auch in anderen Programmen verwendet werden kann.

Schreiben Sie (nachdem Sie Ihr Umfüllprogramm nach Aufteilung in die Module erfolgreich ausgeführt haben) zum Test ein Programm, das die Quadrat- und Kubikzahlen berechnet, die kleiner als einer Konstante `const int N=...;` sind (wählen Sie z. B. `N=250`), und diese in eine Menge einfügt. Geben Sie diese Menge aus.

Zusatzaufgaben für Schnellschreiber: Welche Zahlen kleiner `N` sind sowohl Quadrat- als auch Kubikzahlen? Schreiben Sie dazu ein Programm, das zwei Mengen für Quadrat- und Kubikzahlen anlegt und danach mit der bereits vorhandenen Funktion `isInM` testet, welche Zahlen in beiden Mengen vorhanden sind. Welche sind die drei kleinsten Zahlen, welche die zehnte und 32-ste? In diesem Fall liegen beide Mengen sortiert vor. Wie könnte man mit diesem Wissen die gesuchten Zahlen schneller finden? (Sie können hier zum Testen die Zahlen direkt in das Array der Menge schreiben oder schreiben eine Funktion, die ein Element ohne Aufruf `isInM()` zu der Menge hinzufügt – andernfalls prüfen Sie, welchen Zeitanteil der Aufbau der Mengen und welchen Anteil das Finden der gemeinsamen Elemente beansprucht – dies wird nur bei größeren Werten für `N` sinnvoll möglich sein.)

B.11 Pointer und Rekursion – Stacks, Listen, ...

Auf den ersten Blick haben Pointer und Rekursion nicht viel miteinander zu tun. Wendet man den Blick auf eine der Standarddatenstrukturen, die in der Regel mit Pointern realisiert werden, – den Stack – wird die stackartige Verwaltung der lokalen Variablen bei Funktionsaufrufen und speziell auch bei Rekursion sichtbar.

Rekursion oder Stack? – eine Anwendung: Eine Möglichkeit die Reihenfolge der Ausgabe umzukehren ist, die Daten zunächst auf einen Stack zu legen, und diese dann beim Löschen vom Stack auszugeben. Statt dies explizit auszuprogrammieren kann man auch den Stack der lokalen Variablen bei Funktionsaufrufen benutzen. Vollziehen Sie die Arbeitsweise der folgenden Funktion nach:

```
void bitteUmkehren(void) {
  int zahl;
  cout << "Eine Zahl bitte (Abbruch durch Eingabe der 0): ";
  cin >> zahl;
  if (zahl!=0) {
    bitteUmkehren();
    cout << "Die Zahl davor war die " << zahl << endl;
  }
}
```

Implementieren Sie bei der Ausgabe des Lösungsweges in Ihrem Programm zum Lösen der Umfüll-Probleme die Umkehrung der Gefäßkonfigurationen, sodass diese in der Reihenfolge von Start- zur Zielkonfiguration ausgegeben werden. Schreiben Sie zwei Varianten:

- durch explizite Programmierung eines Stacks
- analog zur obigen Funktion `bitteUmkehren()`

Die Türme von Hanoi: In der Literatur findet man fast ausschließlich den rekursiven Lösungsansatz. Das Umlegen der Scheiben lässt sich aber auch durch einem iterativen Ansatz beschreiben – wir geben ihn hier als Pseudocode an:

```
loeseHanoiIterativ(hoehe)
  initialisiere einen Turm mit <hoehe> Scheiben auf Stab A;
  verschiebe Scheibe 1 von Stab A auf B
  solange nicht zwei Stäbe gleichzeitig leer sind
    finde die zweitgrößte bewegbare Scheibe
    finde den anderen Stab, auf dem oben nicht Scheibe 1 liegt
    verschiebe diese Scheibe auf diesen Stab
```

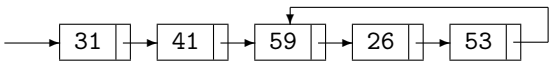
Spielen Sie das Verfahren mit einer `hoehe` von mindestens 4 von Hand durch. Nun soll dies Verfahren in C++ programmiert werden. Jeder Stab wird durch eine Liste realisiert, bei der – wie bei einem Stack – Elemente nur vorne angehängt und vorne entfernt werden (der Anker zeigt also jeweils auf das oberste Element des Stabes). Das Programm soll dabei neben der zu bewegenden Scheibe auch jeweils den Zustand auf den 3 Stäben ausgeben, z. B.:

```
Ich loese die Tuerme von Hanoi mit Hoehe 4.
Startzustand:
A: 4321
B:
C:
Verschiebe Scheibe 1 von A nach B.
A: 432
B: 1
C:
Verschiebe Scheibe 2 von A nach C.
A: 43
B: 1
C: 2
```

```
Verschiebe Scheibe 1 von B nach C.
A: 43
B:
C: 21
Verschiebe Scheibe 3 von A nach B.
A: 4
B: 3
C: 21
Verschiebe Scheibe 1 von C nach A.
A: 41
B: 3
C: 2
...
```

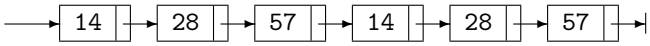
Beachten Sie, dass Sie bei der Ausgabe der Listen deren Inhalt in umgekehrter Reihenfolge ausgeben müssen (Ansatz, siehe oben).

Im Kreis gedreht: Bei unseren bisherigen Listen sind wir stets davon ausgegangen, dass wir beim Anker beginnend über die Verzeigerung entweder – bei einfach verketteten Listen – irgendwann auf den `NULL`-Pointer treffen oder – bei Ringlisten – wieder auf das erste Element. Es sind aber auch beliebige Mischformen denkbar, die zunächst eine Kette von Elementen darstellen, der `next`-Zeiger des letzten Elements dann aber auf ein beliebiges Element der Liste verweist. Sollen die Elemente solch einer Liste ausgegeben werden, kann man nicht einfach dem `next`-Zeiger bis zum `NULL`-Zeiger folgen, sondern man muss die Stelle, an der der Zyklus möglicherweise beginnt, erkennen können. Zu der Liste



könnte die Ausgabe lauten: „`31, 41, Beginn des Zyklus: 59, 26, 53, ...`“ – ist kein Zyklus vorhanden, so werden einfach die Zahlen der Liste ausgegeben. Für die obige Liste ohne den Rückverweis könnte die Ausgabe lauten: „`31, 41, 59, 26, 53, Ende der Liste`“.

Entwerfen Sie eine Funktion, die solche Listen erzeugt, die möglicherweise einen Zyklus enthalten. Schreiben Sie eine weitere Funktion, die den Inhalt solch einer Liste wie oben beschrieben ausgibt. Beachten Sie, dass es dabei auf die Verzeigerung ankommt, nicht auf den Inhalt der Liste! In der Liste



dürfte kein Zyklus erkannt werden. Die Ausgabe lautet dann: „`14, 28, 57, 14, 28, 57, Ende der Liste`“.

Hinweis: Sie sollen die Liste selbst dabei nicht verändern – insbesondere soll in das zugehörige `struct` kein weiterer Selektor hinzugefügt werden.

Analyse einer Listenfunktion: Manchen Funktionen sieht man nicht ohne Weiteres an, welchen Zweck sie erfüllen – fehlende Kommentare erschweren die Analyse zusätzlich. Die folgende Funktion arbeitet auf dem Datentyp

```
struct Liste {
  int value;
  Liste *next;
};
```

Die zu untersuchende Funktion lautet:

```
const Liste* findSomething(const Liste *anker) {
  // was tut diese Funktion? Kommentar vergessen?
  const Liste *fwd, *ffwd; // über den Pointer darf die Liste nicht verändert
  // werden, der Pointer selbst darf verändert werden!
  ffwd=fwd=anker;
  do {
    if (ffwd!=NULL) {
      ffwd=ffwd->next;
      if (ffwd!=NULL) {
        ffwd=ffwd->next;
        fwd=fwd->next;
      }
    }
  } while ((ffwd!=NULL) && (ffwd!=fwd));
  if (ffwd==NULL) {
    return NULL;
  } else {
    ffwd=anker;
    while (fwd!=ffwd) {
      fwd=fwd->next;
      ffwd=ffwd->next;
    }
    return ffwd;
  }
}
```

Im Hauptprogramm könnte dann eine Deklaration einer passenden Variablen und der zugehörige Funktionsaufruf so aussehen: `const Liste *loop=findSomething(anker);`. Beachten Sie nochmals die Regel „lese von rechts nach links“ – `const Liste *` ist ein variabler Pointer auf eine konstante Liste, d. h., über diesen Verweis darf der Inhalt der Liste nicht verändert werden. Der Pointer selbst darf aber verändert werden (er wird hier in irgendeiner Art und Weise zum Durchlaufen der Liste benutzt). Sollte der Zeiger nur auf ein bestimmtes Element zeigen dürfen und unveränderbar sein, so wäre der Datentyp `Liste * const` richtig – ein konstanter Zeiger auf einen (über diesen Zeiger) veränderbaren Listeneintrag. Die Kombination – ein konstanter Zeiger auf einen konstanten Listeneintrag wäre vom Datentyp `const Liste * const` – Pointer können (nicht nur in C/C++) verwirrend sein.

Eine sinnfreie Anwendung: Mit Zahlen lässt sich vieles anstellen, z. B. kann man die Quadrate der einzelnen Ziffern addieren – ein Beispiel: $42 \rightarrow 20$ (da $4^2 + 2^2 = 20$), dieses lässt sich zu Ketten erweitern, z. B. $21 \rightarrow 5 \rightarrow 25 \rightarrow 29 \rightarrow 85 \rightarrow 89 \rightarrow 145 \rightarrow 42 \rightarrow 20 \rightarrow 4 \rightarrow 16 \rightarrow 37 \rightarrow 58 \rightarrow 89 \ldots$ Ausgehend von 2-stelligen Zahlen können die Zwischenergebnisse

nie größer als 162 werden (warum?), sodass solche Zahlenfolgen zwangsweise in Zyklen enden müssen (warum?). Entwerfen Sie ein Programm, das sämtliche Zyklen, die auf diese Weise entstehen können, findet. (Warum können auch bei größeren Startwerten keine weiteren Zyklen entstehen?)

Dieses ist nochmals ein etwas umfangreicheres Programm, an dem Sie die verschiedenen Diagrammtypen und die Umsetzung in C/C++-Code nach der Methode des strukturierten Entwurfs üben können.

Hinweis: Überlegen Sie, ob `Liste` bei dieser Aufgabe ein geeigneter Datentyp ist – vergleichen Sie die Anforderungen mit Ideen aus den anderen Programmen, die im ersten Semester bearbeitet wurden.

Hinweis 2: Um eine Zahl in die einzelnen Ziffern zu zerlegen, schauen Sie sich nochmals das erste Beispielprogramm zu `while`-Schleifen an.

Hinweis 3: Obige sinnfreie Anwendung lässt sich natürlich auch mit anderen Exponenten (also z. B. Kuben statt Quadraten) oder auch zu anderen Basen (nicht nur zur uns geläufigen Basis 10) betrachten. Finden Sie so z. B. alle Zahlen, deren Ziffern hoch drei summiert die Zahl selbst ergeben (u. A. $153 = 1^3 + 5^3 + 3^3$).

C Zusätzliche Übungsaufgaben und alte Klausuraufgaben

Übungsaufgaben (ohne Lösungshinweise):

- **Algorithmus:** Ein Algorithmus ist ein exakt formuliertes Verfahren, das von jedem, der den Text des Algorithmus erhält, auf gleiche Weise durchgeführt werden kann, ohne dass weitere Erläuterungen notwendig sind.

  Ein Computer führt in der Regel Algorithmen aus. Wichtig ist zunächst die Eindeutigkeit des Algorithmus.

  - Ein Anweisung zum Zeichnen eines Dreiecks könnte lauten: *Zeichne nach rechts, zeichne von dort nach links oben, zeichne zum Ausgangspunkt zurück.*

    Ist dies eindeutig? Welche zusätzlichen Informationen fehlen?
  - Speziell bei Programmen sind durch die gewählte Hochsprache die möglichen Einzelschritte vorgegeben. Genauso können wir bei Algorithmen vorgeben, dass wir z. B. nur Lineal und Zirkel verwenden wollen. Entwerfen Sie Algorithmen zum Zeichnen
    * eines gleichseitigen Dreiecks
    * eines rechtwinkligen Dreiecks, bei dem die Hypotenuse parallel zum (unteren) Blattrand liegt. (Hinweis: hier ist die genaue Lage des rechten Winkels offen gelassen.)

- **Quadratische Gleichungen** der Form $ax^2 + bx + c = 0$ lassen sich leicht mit der Mitternachtsformel lösen. Entwerfen Sie einen Algorithmus, der zu den Koeffizienten a, b und c bestimmt, ob es zwei, eine oder keine Lösung gibt und das Ergebnis dem Benutzer mitteilt.

  Geben Sie den Entwurf einmal als Struktogramm und einmal als Programmablaufplan an.

  Setzen Sie Ihren Entwurf in ein C(++)-Programm um. Ihr `main()` soll dabei lediglich aus den zwei Aufrufen zur Eingabe sowie Berechnung (mit integrierter Ausgabe) bestehen.

- **Studienstiftung:** Herr S. Pendabel hat eine Stiftung zur Förderung von begabten Studierenden eingerichtet. Dazu wurde ein Betrag von 1000000 Euro langfristig mit festem Zinssatz angelegt. Von den Erlösen sollen jedes Jahr 5 Studierende mit zu Beginn je 600 Euro monatlich gefördert werden. Um die Inflation auszugleichen, erhöht sich dieser Betrag jährlich zum 1.1. um einen festen Prozentsatz. Um sich für die Stiftung zu bewerben, soll man ein Programm entwerfen, das die Entwicklung des Vermögens der Stiftung simuliert, und abhängig von den Zinssätzen berechnet, in welchem Jahr entweder das Vermögen aufgebraucht ist oder wann es sich verdoppelt hat (dann können wir das Vermögen aufteilen und eine zweite Stiftung ins Leben rufen).

  Geben Sie den Entwurf als Struktogramm und Programmablaufplan (mit Daten) an, setzen Sie Ihr Programm in C(++) um und testen Sie es u.A. mit einem Zinssatz von 4% und einem Inflationsausgleich von 2%.

- **Römische Zahlen:** Im römischen Reich wurde ein anderes Zahlsystem verwendet. Die alten Römer verwendeten als Zahlzeichen I (Wert 1), V (=5), X (=10), L (=50), C (=100), D (=500) und M (=1000). Begonnen wird links mit dem Symbol der größten Zahl. Dabei werden die Symbole I, X, C, M höchstens drei Mal hintereinander geschrieben. Die

Symbole V, L, D kommen nur einzeln vor. Der Wert eines Symbols einer kleineren Zahl, die vor dem einer größeren steht, wird von diesem subtrahiert (es darf nur höchstens ein kleineres Symbol einem größeren vorangestellt werden). Folgende Regeln sind weiterhin zu beachten: Einer können nur von Fünfern und Zehnern abgezogen werden. Zehner können nur von Fünfzigern und Hundertern abgezogen werden. Hunderter können nur von Fünfhundertern und Tausendern abgezogen werden.

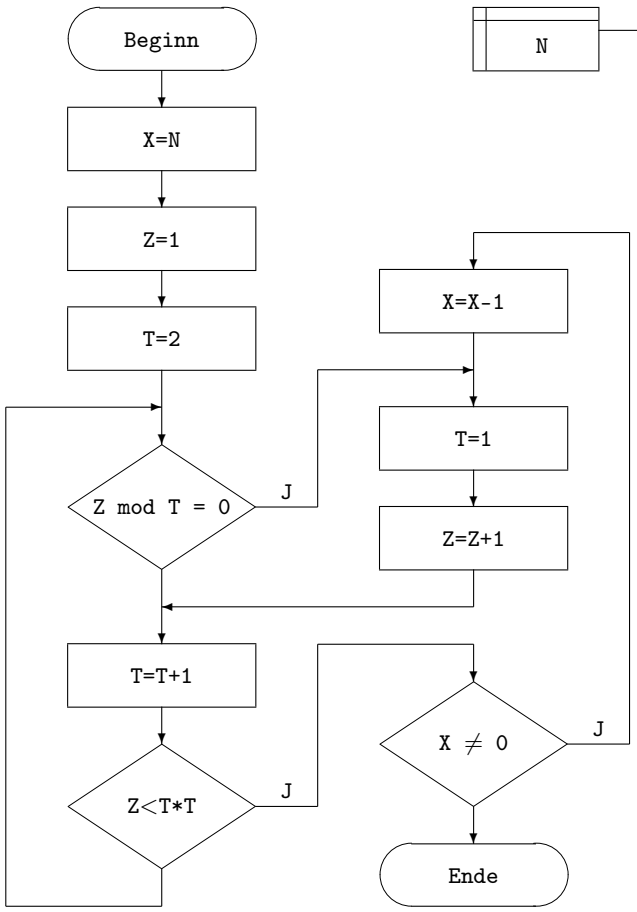
Beschreiben Sie einmal mit Hilfe der EBNF und einmal mit Syntaxdiagrammen den formalen Aufbau der Römischen Zahlen (es können nur die Zahlen kleiner 4000 dargestellt werden).

Konstruieren Sie einen Algorithmus, der zu einer Römischen Zahl (<4000) die Zahl 1 hinzuaddiert. Geben Sie Ihren Entwurf als Struktogramm und Programmablaufplan an, geben Sie ggf. auch ein Programmhierarchiediagramm mit an. (Eine Umsetzung in ein C(++)-Programm ist nicht nötig.)

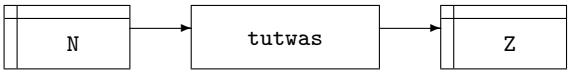
- **Unstrukturierte Entwürfe** sind nicht nur schwer in Programme umzusetzen, sie sind meist auch nur schlecht nachvollziehbar. Sinnvolle Wartung und Veränderungen sind fast nie möglich, da die Nebeneffekte kaum abzuschätzen sind. Zur Abschreckung betrachten wir folgenden Entwurf (der Algorithmus tut tatsächlich etwas sinnvolles).

  Was berechnet der Algorithmus? Analysieren Sie die Idee und geben Sie einen strukturierten Entwurf an, der die ursprüngliche Idee soweit möglich beibehält.

PA:



DF:



- **Minimale Differenz:** Gegeben seien 6 paarweise verschiedene Ziffern zwischen 0 und 9. Diese sind so in zwei 3-stellige Zahlen aufzuteilen, dass die Differenz zwischen diesen beiden Zahlen möglichst gering ist. Beispiele: aus den Ziffern 1, 3, 4, 6, 8, 9 lassen sich die Zahlen 398 und 416 bilden, die minimale Differenz ist 18; aus den Ziffern 0, 1, 2, 4, 5, 6 lassen sich die Zahlen 165 und 204 bilden, die minimale Differenz ist 39 (alternative Lösung: 462 und 501, es reicht in solchen Fällen 1 Lösung anzugeben).

  – Entwerfen (Struktogramm und PA, PH, DH, DF) Sie einen Algorithmus (und setzen diesen dann in ein C(++)-Programm um), der 6 Ziffern in aufsteigender Reihenfolge einliest, aus diesen dann zwei dreistellige Zahlen bildet, deren Differenz möglichst gering ist, und diese Zahlen sowie die Differenz ausgibt.

    Gliedern Sie das Hauptprogramm dabei in eine Eingabe-Prozedur, eine Berechne2Zahlen-Funktion und die Ausgabe.

  – Überlegen und erläutern Sie, ob (und wenn ja wie) Sie das Programm ändern müssten, wenn auch gleiche Zahlen in der Eingabe erlaubt sind.

- **Simulation einer Ampelsteuerung:** An einer T-Kreuzung treffen Autos aus 3 Richtungen (W(est), O(st) uns S(üd)) aufeinander. Um die vielzähligen Unfälle an dieser Kreuzung zu vermeiden, soll eine Ampel aufgestellt werden (aus jeder Richtung getrennt für jede der zwei möglichen Richtungen). Zu Beginn seien die Ampeln für W→O und O→W grün, gleichzeitig kann ohne Gefahr auch W→S grün geschaltet werden. Danach sollen reihum jeweils die Fahrzeuge aus allen Richtungen grün bekommen. Dazu müssen andere dann auf rot umschalten. Es sollen dabei möglichst wenige Ampeln umschalten, bis ein kompletter Zyklus um ist (d. h., bis wieder die Ampeln vom Beginn grün sind). Nach W→O, O→W und W→S grün könnte W→O, O→W auf geld und dann rot umschalten, W→S auf grün bleiben und S→W sowie S→O auf rot/gelb und dann grün umschalten.

  Modellieren Sie dieses Szenario zu Ende und schreiben ein C(++)-Programm, das diesen Zyklus inklusiver aller Ampelphasen beschreibt – dabei sollen jeweils die umzuschaltenen Ampeln angegen werden und direkt im Anschluss, welche Verbindungen nun jeweils freie Fahrt haben.

Alte Klausuraufgaben (Lösungshinweise finden Sie unter `http://www.inf4dhbw.de.vu`):

Addition großer Zahlen: C++ verwendet bei ganzen Zahlen 16 Bit für `short int`, 32 Bit für `int` und 64 Bit für `long int`. Der damit darstellbare Zahlenbereich ist für die meisten Anwendungen ausreichend. Will man längere Zahlen verarbeiten, so muss man die einzelnen Stellen der Zahl in einer Liste darstellen und diese dann stellenweise verarbeiten – wir werden in dieser Aufgabe stets 3 Dezimalstellen zusammenfassen und große Zahlen als Liste von solchen 3er-Blöcken darstellen.

Beispiele:

`2098067145` wird dargestellt als `(2,98,67,145)`
`5102952000` wird dargestellt als `(5,102,952,0)`

Die Addition geht nun stellenweise vor wie beim schriftlichen Addieren:

```
  2 098 067 145
+ 5 102 952 000
===============
  7 201 019 145
```

Das Ergebnis ist also die Liste `(7,201,19,145)`. Beachten Sie den Übertrag, der hierbei vorkommt (`67+952=1019` ⇒ Ergebnis für diesen 3er-Block ist `19` und der Übertrag muss bei dem links davon stehenden 3er-Block berücksichtigt werden).

1. Erstellen Sie ein Datenhierarchiediagramm (DH) für die hier verwendete Liste und geben Sie die Datenstruktur in C++ an (die maximale Anzahl der 3er-Blöcke soll durch eine Konstante `Max3Block` definiert sein).
2. Entwerfen Sie einen Algorithmus zur Addition großer Zahlen und geben Sie den zugehörigen Programmablaufplan (PA) und das Programmnetz (PN) an. Ist das Ergebnis zu groß, so soll eine Fehlermeldung ausgegeben werden.
3. Setzen Sie Ihren Entwurf in eine C++-Funktion um.

Ausgabe großer Zahlen: C++ verwendet bei ganzen Zahlen 16 Bit für `short int`, 32 Bit für `int` und 64 Bit für `long int`. Der damit darstellbare Zahlenbereich ist für die meisten Anwendungen ausreichend. Will man längere Zahlen verarbeiten, so muss man die einzelnen Stellen der Zahl in einer Liste darstellen und diese dann stellenweise verarbeiten – wir werden in dieser Aufgabe stets 3 Dezimalstellen zusammenfassen und große Zahlen als Liste von solchen 3er-Blöcken darstellen.

Beispiele:

`2098067145` wird dargestellt als `(2,98,67,145)`
`5102952000` wird dargestellt als `(5,102,952,0)`

Wir wollen hier nun eine Funktion zur Ausgabe solcher Zahlen schreiben, da dies mit den Ausgabefunktionen `cout` und `printf` nicht direkt möglich ist. Achten Sie dabei auf die Nullen! Es sollen keine führenden Nullen ausgegeben werden!

Beispiele:

`(2,98,67,145)` wird ausgegeben als `2098067145`
`(0,0,23,0)` wird ausgegeben als `23000`
`(0,0,0,0)` wird ausgegeben als `0`

1. Entwerfen Sie einen Algorithmus zur Ausgabe großer Zahlen und geben Sie den zugehörigen Programmablaufplan (PA) und das Programmnetz (PN) an. (Hinweis: Fehlerhafte Zahlen (z. B. 3er-Blöcke mit Werten größer 1000) müssen nicht erkannt werden. Geben Sie die Diagramme für PA und PN getrennt an oder markieren Sie eindeutig, welche Teile zum PA und welche zum PN gehören. PA und PN sollen hier so detailliert sein, dass eine Umsetzung in eine C++-Funktion 1:1 möglich ist.)
2. Setzen Sie Ihren Entwurf in eine C++-Funktion `void put (const myint x)` um, die als Parameter eine *große Zahl* übergeben bekommt. Die Datenstruktur für die großen Zahlen ist ein Feld aus `integer`-Werten, wobei Sie hier davon ausgehen können, dass für die einzelnen Werte nur die Zahlen 0 bis 999 vorkommen.

```
const int Max3Block = 4;
typedef int myint[Max3Block];

void put (const myint x)
{

  // fügen Sie hier Ihren Code ein

}
```

Phythagoräische Zahlentripel `(a,b,c)` haben die Eigenschaft, dass `a*a+b*b=c*c` gilt, Beispiele: `(3,4,5)`, `(5,12,13)`, `(6,8,10)` sind Phythagoräische Zahlentripel, `(5,3,4)` und `(1,2,3)` sind keine.

Schreiben Sie eine C++-Funktion `void phyth(const int n)`, die alle Phythagoräischen Zahlentripel `(a,b,c)` mit der Eigenschaft `a*a+b*b=c*c` und `1<a<b<c<n` berechnet und ausgibt. Die Zahlentripel sollen in aufsteigender Reihenfolge bzgl. des Wertes `c` angegeben werden. Für `n=14` soll die Ausgabe somit

```
(3,4,5)
(6,8,10)
(5,12,13)
```

lauten. Achten Sie auch darauf, dass kein Zahlentripel doppelt ausgegeben wird. (Hinweis: es ist möglich die Aufgabe ohne Verwendung eines Sortieralgorithmus zu lösen! Funktionen zum Einlesen des Wertes `n` oder `int main()` sind nicht erforderlich.)

Beschreiben Sie anschließend, wie Sie Ihr Programm ändern müssten, wenn die Zahlentripel in aufsteigender Reihenfolge bzgl. des Wertes `a` angegeben werden sollen.

Hotel Confusio: Ein Freund erzählt Ihnen, dass er ein Hotel geerbt hat und dass dieses 23 Zimmer mit 37 Betten hätte. Später überlegen Sie, wieviel Einzel- und wieviel Doppelzimmer das Hotel wohl hat und nach etwas raten kommen Sie auf die Lösung, dass es 9 Einzelzimmer und 14 Doppelzimmer sein müssten.

Um in Zukunft auf solche Situationen besser vorbereitet zu sein, beschließen Sie eine C/C++-Funktion zu entwerfen, die als Eingangsparameter die Anzahl der Zimmer und Betten bekommt und als Ausgangsparameter die Anzahl der Einzel- und Doppelzimmer berechnet. Ferner soll die Funktion über eine ebenfalls zu übergebene `bool`-Variable den Wert `true` liefern, wenn die Berechnung fehlerfrei möglich ist, und `false`, falls die Eingaben zu keiner gültigen Lösung führen können (z. B. muss die Anzahl der Betten immer mindestens so groß wie die der Zimmer sein).

Geben Sie das Datenflussdiagramm (DF) der Ebene 0 an (Funktion als ein Kästchen), sowie einen Programmablaufplan (PA) für die von Ihnen entworfene Funktion mit den nötigen Verfeinerungen. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich wäre, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: die Parameter sind an die Funktion zu übergeben und werden von dieser zurückgeliefert, Sie müssen sich *nicht* um die Ein- und Ausgabe der Werte kümmern. Eine Umsetzung in C/C++ ist hier *nicht* gefordert.

Spiegelzahlen: Eine Spiegelzahl ist eine Zahl, die von vorne nach hinten gelesen identisch mit der Zahl ist, die von hinten nach vorne gelesen wird. Somit sind 343, 1001 und 7 Spiegelzahlen, 62263 und 42 jedoch nicht.

Geben Sie zunächst EBNF-Regeln an, die genau solche Spiegelzahlen erzeugen. (Zusatzaufgabe: Beachten Sie dabei, dass außer der 0 keine Spiegelzahl am Anfang und Ende Nullen hat.)

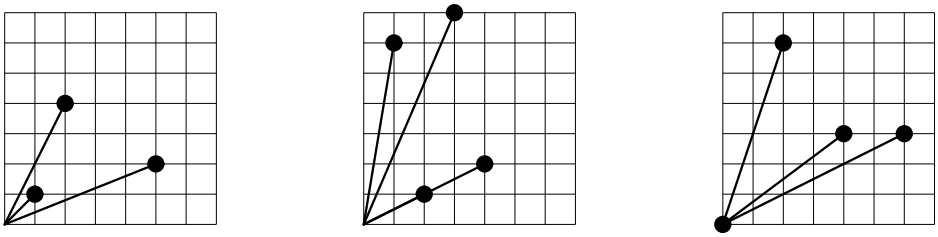
Für eine Umsetzung in eine C/C++-Funktion liegt bereits folgendes Programmfragment vor:

```
typedef int ziffern[10]; // ein unsigned int hat maximal 10 Ziffern
void gespiegelt(const unsigned int n, bool & erg) {
   ziffern zehnziffern;
   int2ziffern(n,zehnziffern);
   spiegeltest(zehnziffern,erg);
}
```

Die Idee ist also, zunächst die als `unsigned int` übergebene Zahl mit der Funktion `int2ziffern` in ihre einzelnen Stellen zu zerlegen (bei 343 also in 3, 4 und 3), diese Stellen in das Array `zehnziffern` zu speichern (ggf. den Rest des Arrays `zehnziffern` mit Nullen zu füllen) und danach diese einzelnen Stellen mit der Funktion `spiegeltest` auf geeignete Weise miteinander zu vergleichen und dabei die Variable `erg` auf `true` oder `false` zu setzen. Führen Sie den Entwurf zu Ende und setzen diesen dann in die entsprechenden C/C++-Funktionen um. Das Hauptprogramm `int main()` muss nicht geschrieben werden, ebenso brauchen Sie sich nicht um das Einlesen der Zahl `n` oder die Ausgabe des Ergebnisses zu kümmern.

Gefordert sind die Programmablaufpläne mit Daten (PAD) *oder* Programmnetze (PN) der Funktionen `gespiegelt`, `int2ziffern` und `spiegeltest`, ein Programmhierarchiediagramm (PH), sowie die Umsetzung von `int2ziffern` und `spiegeltest` in C/C++.

Kleinster Winkel: Gegeben ist eine Menge von Punkten $(x, y) \in \mathbb{N}_0 \times \mathbb{N}_0$ in der Ebene. Für jeden dieser Punkte (x, y) bildet die Gerade vom Ursprung $(0, 0)$ zu diesem Punkt (x, y) einen Winkel mit der x-Achse. Ihre Aufgabe ist, eine Funktion zu entwerfen, die aus einer Menge von gegebenen Punkten den Punkt bestimmt, für den dieser Winkel am kleinsten ist, und diesen als Ergebnis der Funktion zurückgibt (keine Ausgabe auf dem Bildschirm). Wenn bei mehreren Punkten dieser Winkel identisch ist, so soll derjenige zurückgegeben werden, der dem Ursprung am nächsten ist. Ist ein Punkt der Ursprung, so soll stets dieser zurückgegeben werden. (Hinweis: Sie brauchen keine trigonometrischen Funktionen, überlegen Sie sich, wie die Steigungen der Geraden mit dem Winkel zusammenhängen.)



Im ersten Beispiel ist das Ergebnis $(5, 2)$, im zweiten $(2, 1)$ (gleicher Winkel wie $(4, 2)$, aber näher am Ursprung), im dritten ist das Ergebnis $(0, 0)$ (da der Ursprung als Punkt dabei ist).

Geben Sie die verwendeten Datenstrukturen als Datenhierarchiediagramm (DH) an. Entwerfen Sie eine Funktion, die ein Array mit Punkten übergeben bekommt und die Koordinaten des

gesuchten Punktes zurückgibt. Geben Sie ein Struktogramm für die von Ihnen entworfene Funktion an. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich ist, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: die Parameter sind an die Funktion zu übergeben und werden von dieser zurückgeliefert, Sie müssen sich *nicht* um die Ein- und Ausgabe der Werte kümmern.

Setzen Sie danach Ihren Entwurf in C/C++ um.

Als Quersumme einer Zahl `n` bezeichnet man die Summe der Ziffern der Zahl `n`.

Entwerfen Sie eine Funktion `quer`, die als Eingangsparameter eine Zahl `n` (Datentyp `unsigned int`) erhält und in der ebenfalls zu übergebenen Variablen `erg` (Datentyp auch hier `unsigned int`) die Quersumme von `n` berechnet.

Beispiele: Quersumme von 19092008 ist 29 (Hinweis: wir betrachten hier nicht die iterierte Quersumme, die oben zum Ergebnis 2 führen würde (2+9=11 und schließlich 1+1=2)). Quersumme von 102936 ist 21.

Geben Sie das Datenflussdiagramm (DF) der Ebene 0 an (Funktion als ein Kästchen), sowie einen Programmablaufplan mit Daten (PAD) (wahlweise ist statt dem PAD auch ein Programmnetz (PN) erlaubt) für die von Ihnen entworfene Funktion mit den nötigen Verfeinerungen. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich ist, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: die Parameter sind an die Funktion zu übergeben und werden von dieser zurückgeliefert, Sie müssen sich *nicht* um die Ein- und Ausgabe der Werte kümmern.

Setzen Sie anschließend Ihren Entwurf in eine C/C++-Funktion um (nur die Funktion `quer`). Das Hauptprogramm `int main()` muss nicht geschrieben werden, ebenso brauchen Sie sich nicht um das Einlesen der Zahl `n` oder die Ausgabe des Ergebnisses zu kümmern.

Mit EBNF lassen sich auch Zahlen mit bestimmten Eigenschaften erzeugen: Eine Zahl ist genau dann durch 3 teilbar, wenn die Quersumme durch 3 teilbar ist. Ist die erste Ziffer z. B. eine 4 und soll die ganze Zahl durch 3 teilbar sein, so muss die Summe der restlichen Ziffern durch 3 geteilt den Rest 2 ergeben (z. B. 468).

Geben Sie EBNF-Regeln an, die genau die Dezimalzahlen erzeugen, die durch 3 teilbar sind. (Wer dabei beachtet, dass außer der Zahl 0 keine Zahl führende Nullen hat, erhält 2 Zusatzpunkte.)

Bruchrechnung ist immer wieder ein Problem. Gegeben sei der Datentyp

```
struct Bruch{
  int zaehler;
  unsigned int nenner;
};
```

Entwerfen Sie eine Funktion `kleiner`, die als Eingangsparameter zwei Brüche `b1` und `b2` erhält und in der ebenfalls zu übergebenen `bool`-Variablen den Wert `true` liefert, wenn der Wert des Bruchs `b1` kleiner oder gleich dem Wert des Bruchs `b2` ist, und `false` sonst. Ist der `nenner`

eines Bruchs gleich 0, so soll der Wert des Bruchs hier stets als 0 betrachtet werden (ist der `zaehler` 0, gilt dies sowieso).

Beispiele: ist `b1=(3,5)` und `b2=(2,3)`, so hat nach Aufruf von `kleiner(b1,b2,erg)` die Variable `erg` den Wert `true`; ist `b1=(3,5)` und `b2=(0,1)`, so hat nach Aufruf von `kleiner(b1,b2,erg)` die Variable `erg` den Wert `false`; ist `b1=(6,3)` und `b2=(2,1)`, so hat nach Aufruf von `kleiner(b1,b2,erg)` die Variable `erg` den Wert `true` (da 2=2); ist `b1=(7,0)` und `b2=(0,3)`, so hat nach Aufruf von `kleiner(b1,b2,erg)` die Variable `erg` den Wert `true` (da `b1` und `b2` als 0 interpretiert werden).

Geben Sie das Datenhierarchiediagramm (DH) für den Datentyp `Bruch` an, sowie einen Programmablaufplan (PA) für die von Ihnen entworfene Funktion mit den nötigen Verfeinerungen. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich ist, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: die Parameter sind an die Funktion zu übergeben und werden von dieser zurückgeliefert, Sie müssen sich *nicht* um die Ein- und Ausgabe der Werte kümmern.

Setzen Sie anschließend Ihren Entwurf in eine C/C++-Funktion um (nur die Funktion `kleiner`). Das Hauptprogramm `int main()` muss nicht geschrieben werden, ebenso brauchen Sie sich nicht um das Einlesen der Brüche oder die Ausgabe des Ergebnisses zu kümmern.

Quadrat- und Kubikzahlen: Entwerfen Sie eine Funktion `aufvier`, die zu einer gegebenen Zahl `n` alle positiven ganzen Zahlen `<=n` in aufsteigender Reihenfolge ausgibt, die entweder Quadratzahl oder Kubikzahl sind. Ist eine Zahl sowohl Quadrat- als auch Kubikzahl (z. B. ist $64 = 8^2 = 4^3$), so soll diese nur einmal ausgegeben werden. Der Aufruf `aufvier(120)` soll so z.,B. die Zahlen `1 4 8 9 16 25 27 36 49 64 81 100` in dieser Reihenfolge ausgeben.

Geben Sie ein Struktogramm für die von Ihnen entworfene Funktion an. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich ist, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: der Parameter `n` wird an die Funktion übergeben, Sie müssen sich *nicht* um die Eingabe des Wertes `n` kümmern.

Setzen Sie danach Ihren Entwurf in C/C++ um (nur die Funktion `aufvier`, das Hauptprogramm `int main()` muss nicht geschrieben werden).

Hinweis: Es ist hier kein Sortieralgorithmus gefordert. Sie benötigen in dieser Aufgabe keine Arrays. Es reichen neben dem Eingangsparameter `n` zwei weitere Variablen aus (Sie dürfen aber auch mehr Variablen verwenden).

Gerade Zahlen: Entwerfen Sie ein kleines Programm, das von einem Benutzer Zahlen einliest und dabei zählt, wieviel gerade Zahlen eingelesen wurden. Das Ende erfolgt durch eine Eingabe einer Zahl kleiner oder gleich 0 durch den Benutzer. Das Programm soll danach ausgeben, wieviel gerade Zahlen vom Benutzer eingegeben wurden.

Beispiel: Der Benutzer gibt folgende Zahlen ein (jeweils getrennt durch die `<RETURN>`-Taste): `42 11 7 21 18 4 3 -2`, dann soll die Ausgabe lauten:
`Es wurden 3 gerade Zahlen eingegeben.`

Geben Sie Ihren Entwurf als Struktogramm an. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich wäre, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: Eine Umsetzung in C/C++ ist hier *nicht* gefordert.

Zahlensummen: Entwerfen Sie ein kleines Programm, das von einem Benutzer Zahlen einliest und diese summiert, bis diese Summe exakt 42 beträgt. Gibt der Benutzer eine Zahl ein, so dass die Summe größer 42 würde, so soll diese Zahl in der Summe ignoriert werden. Das Programm soll danach ausgeben, wieviel Zahlen vom Benutzer eingegeben wurden.

Beispiel: Der Benutzer gibt folgende Zahlen ein (jeweils getrennt durch die `<RETURN>`-Taste): `20 15 10 5 3 2`, dann soll die Ausgabe lauten:
`Es wurden 6 Zahlen eingegeben.` (20+15=35, die 10 wird zwar bei der Anzahl berücksichtigt, aber in der Summe ignoriert, da 35+10 größer 42 ist, 35+5=40, die 3 wird nochmals in der Summe ignoriert, Abbruch nach Eingabe der 2, da 40+2=42.)

Geben Sie Ihren Entwurf als Struktogramm an. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich wäre, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: Eine Umsetzung in C/C++ ist hier *nicht* gefordert.

Ausgewürfelt: In dieser Aufgabe ist eine Funktion `markiere_max(.,.)` innerhalb eines Spiels zu entwerfen und in C- oder C++-Code umzusetzen.

Die Funktion soll dabei zwei Arrays als Parameter bekommen: Das eine Array enthält `int`-Werte (die Werte seien hier aus dem Bereich 1 bis 6), ein zweites Array enthält Wahrheitswerte, die angeben, ob der Wert im `int`-Array zu berücksichtigen ist. Die Funktion soll aus dem ersten Array den größten zu berücksichtigenden Wert heraussuchen und dann das Wahrheitswerte-Array so modifizieren, dass danach nur noch genau die Stellen wahr sind, die zuvor auch wahr waren und dessen zugehöriger Wert dem zuvor bestimmten Maximalwert entsprechen. (Dies könnte z. B. ein wiederholtes Würfeln in einem Spiel simulieren, bei dem die Spieler nach und nach ausscheiden (nur, wer im zweiten Array markiert ist, ist noch am Spiel beteiligt).)

Gegeben sind die C/C++-Definitionen:

```
const int MAX_SPIELER=7;
typedef int werte[MAX_SPIELER];
typedef bool ist_dabei[MAX_SPIELER];
```

Ein Beispiel: seien zwei Arrays wie folgt definiert:

```
werte     zahlen  = {3,4,5,2,4,1,4};
ist_dabei spieler = {0,1,0,1,1,1,0}; // true entspricht 1, false entspricht 0
```

Dann soll der Aufruf `markiere_max(zahlen,spieler)` das Array `spieler` so verändern, dass es danach die Werte `0,1,0,0,1,0,0` enthält (4 ist der größte zu berücksichtigende Wert (`zahlen[2]` ist zwar 5, aber `spieler[2]` ist 0) und nur an den Indizes 1 und 4 war zuvor `spieler[.]` als wahr markiert und `zahlen[.]` gleich 4). Sind alle Einträge im Array `spieler[]` gleich 0, so soll die Ausgabe „`Keiner mehr dabei!`“ erfolgen.

Geben Sie das Datenflussdiagramm (DF) der Ebene 0 an (Funktion als ein Kästchen), sowie einen Programmablaufplan mit Daten (PAD) oder ein Programmnetz (PN) für die von Ihnen entworfene Funktion mit den nötigen Verfeinerungen. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich ist, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: die Parameter sind an die Funktion zu übergeben und werden von dieser zurückgeliefert, Sie müssen sich *nicht* um die Ein- und Ausgabe der Werte kümmern. Geben Sie schließlich noch die Umsetzung der Funktion in C/C++ an.

Ausgewürfelt (Variante): In dieser Aufgabe ist eine Funktion `ausgabe_max(.,.)` innerhalb eines Spiels zu entwerfen und in C- oder C++-Code umzusetzen.

Die Funktion soll dabei zwei Arrays als Parameter bekommen: Das eine Array enthält `int`-Werte (die Werte seien hier aus dem Bereich 1 bis 6), ein zweites Array enthält Wahrheitswerte, die angeben, ob der Wert im `int`-Array zu berücksichtigen ist. Die Funktion soll aus dem ersten Array den größten zu berücksichtigenden Wert heraussuchen und die Indizes ausgeben, deren zugehöriger Wert dem zuvor bestimmten Maximalwert entsprechen. (Dies könnte z. B. ein wiederholtes Würfeln in einem Spiel simulieren, bei dem die Spieler nach und nach ausscheiden (nur, wer im zweiten Array markiert ist, ist noch am Spiel beteiligt).)

Gegeben sind die C/C++-Definitionen:

```
const int MAX_SPIELER=7;
typedef int werte[MAX_SPIELER];
typedef bool ist_dabei[MAX_SPIELER];
```

Ein Beispiel: seien zwei Arrays wie folgt definiert:

```
werte     zahlen  = {3,4,5,2,4,1,4};
ist_dabei spieler = {0,1,0,1,1,1,0}; // true entspricht 1, false entspricht 0
```

Dann soll der Aufruf `ausgabe_max(zahlen,spieler)` die Indizes `1` und `4` ausgeben (4 ist der größte zu berücksichtigende Wert (`zahlen[2]` ist zwar 5, aber `spieler[2]` ist 0, ebenso ist `6` nicht auszugeben, da `spieler[6]` 0 ist). Sind alle Einträge im Array `spieler[]` gleich 0, so soll die Ausgabe „`Keiner mehr dabei!`“ erfolgen.

Geben Sie das Datenflussdiagramm (DF) der Ebene 0 an (Funktion als ein Kästchen), sowie einen Programmablaufplan mit Daten (PAD) oder ein Programmnetz (PN) für die von Ihnen entworfene Funktion mit den nötigen Verfeinerungen. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich ist, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: die Parameter sind an die Funktion zu übergeben und werden von dieser zurückgeliefert, Sie müssen sich *nicht* um die Ein- und Ausgabe der Werte kümmern. Geben Sie schließlich noch die Umsetzung der Funktion in C/C++ an.

Datenstrukturen: In dieser Aufgabe ist eine **Datenstruktur** zu entwerfen, als Datenhierarchiediagramm darzustellen und in C/C++-Code umzusetzen. Die Datenstruktur soll zur Verwaltung der Studierenden dienen und soll dabei die Daten für einen Studierenden beinhalten. Folgende Daten sollen gespeichert werden: Vorname, Nachname, Matrikelnummer und das Geburtsdatum (speichern Sie für das Datum jeweils Zahlen für den Tag, Monat und das Jahr). Das Datum soll dabei ein eigener Datentyp sein.

Geben Sie ein Datenhierarchiediagramm für Ihre Datenstruktur an und geben Sie die Umsetzung in C/C++-Code an.

Eine weitere **Datenstruktur**: Die Datenstruktur soll zur Verwaltung der Vorlesungen dienen und soll dabei die Daten für eine Vorlesung beinhalten. Folgende Daten sollen gespeichert werden: Titel und Dauer (in Minuten) der Vorlesung, Vor- und Nachname des Dozenten.

Geben Sie ein Datenhierarchiediagramm für Ihre Datenstruktur an und geben Sie die Umsetzung in C/C++-Code an.

Wissensfrage: Beschreiben Sie in 4 kurzen Sätzen, was ein Programm von einem Algorithmus unterscheidet.

Wissensfrage: Geben Sie mindestens 3 gute Gründe an, warum eine Aufteilung des Source-Codes eines Projektes in einzelne Module sinnvoll ist (kurze Begründungen reichen jeweils).

EBNF/Syntaxdiagramm: Die `switch`-Anweisung wird gerne für große Fallunterscheidungen benutzt: ein Beispiel-Code (`i` sei eine `int`-Variable):

```
switch(i) {
  case 17: cout << "Oh, eine Primzahl";
  case 31: cout << "Was für eine schöne Primzahl"; break;
  case 91: cout << "Das ist doch keine Primzahl"; break;
  default: cout << "Was soll das denn?";
}
```

Beschreiben Sie die Syntax der `switch`-Anweisung als EBNF oder als Syntaxdiagramm. Treten dabei Nichtterminale für Variablennamen, Typnamen, Arithmetische Ausdrücke, Boolesche Ausdrücke oder Kommandos auf, so müssen Sie diese *nicht* weiter spezifizieren.

Verbunde und eigene Datentypen: Was bewirken jeweils die folgenden Zeilen C/C++-Code? (1–2 Sätze reichen jeweils)

- Variante 1:

  ```
  struct Datum {
    int tag;
    int monat;
    int jahr;
  };
  ```

- Variante 2:

  ```
  struct {
    int tag;
    int monat;
    int jahr;
  } Datum;
  ```

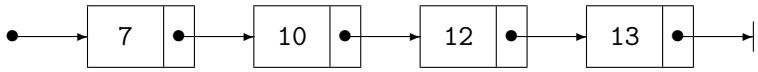
- Variante 3:

  ```
  typedef struct {
    int tag;
    int monat;
    int jahr;
  } Datum;
  ```

Einfach verkettete Listen: In dieser Aufgabe beschäftigen wir uns mit einfach verketteten Listen, der Anker zeige dabei auf das erste Element.

1. Geben Sie eine C- oder C++-Typdefinition für eine einfach verkettete Liste aus `int`-Zahlen an. Der Datentyp für den Anker soll dabei `intListenPtr` heißen.

2. Schreiben Sie eine C/C++-Funktion `void arithm(intListenPtr liste)`, die als Parameter den Anker auf eine einfach verkettete Liste bekommt. Ist die Liste leer, so soll die Meldung „`Liste ist leer.`“ ausgegeben werden, ansonsten das arithmetische Mittel (Summe der Elemente geteilt durch deren Anzahl). Zu der Liste

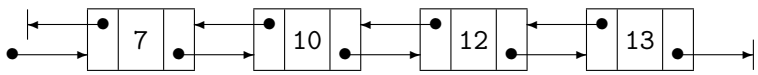
   soll die Ausgabe lauten: „`Das arithmetische Mittel beträgt 10.5`“

Beachten Sie: die Parameter sind an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe der Werte kümmern.

Doppelt verkettete Listen: In dieser Aufgabe beschäftigen wir uns mit doppelt verketteten Listen, der Anker zeige dabei auf das erste Element. Bei doppelt verketteten Listen hat jedes Listenelement sowohl einen Zeiger auf das nachfolgende als auch auf das vorhergehende Element der Liste.

1. Geben Sie eine C- oder C++-Typdefinition für eine doppelt verkettete Liste aus `int`-Zahlen an. Der Datentyp für den Anker soll dabei `intListenPtr` heißen.

2. Schreiben Sie eine C/C++-Funktion `void loesche(intListenPtr & liste, const int i)`, die als Parameter den Anker auf eine doppelt verkettete Liste und eine ganze Zahl bekommt. Hat die Liste weniger als `i` Elemente, so soll die Meldung „`Liste ist kürzer.`“ ausgegeben werden, ansonsten soll das `i`-te Element aus der Liste gelöscht werden. Zu der Liste

   (der Pointer `anker` zeige auf die `7`) soll beim Aufruf `loesche(anker,3)` die `12` entfernt werden – beim Aufruf `loesche(anker,5)` soll die Ausgabe lauten: „`Liste ist kürzer.`“

   Hinweis: Beachten Sie beim Löschen die Randfälle! Das zu löschende Element könnte keinen Nachfolger und/oder keinen Vorgänger haben.

3. Erläutern Sie kurz (1–2 Sätze genügen), für welchen Fall der Parameter `liste` als Referenzparameter („`&`“) übergeben werden muss.

Beachten Sie: die Parameter sind an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe der Werte kümmern.

Kleine und große Zahlen: Entwerfen Sie ein kleines Programm, das von einem Benutzer Zahlen einliest und sich dabei merkt, welches die kleinste und welches die größte der eingelesenen Zahlen war. Das Ende erfolgt durch eine Eingabe der Zahl 0 durch den Benutzer. Das Programm soll danach die kleinste und größte eingegebene Zahl ausgeben.

Beispiel: Der Benutzer gibt folgende Zahlen ein (jeweils getrennt durch die `<RETURN>`-Taste): `42 11 7 21 18 4 3 -2 0`, dann soll die Ausgabe lauten:
`Die kleinste Zahl war die -2, die größte Zahl war die 42.`

Beispiel 2: Der Benutzer gibt folgende Zahlen ein (jeweils getrennt durch die `<RETURN>`-Taste): `-21 0`, dann soll die Ausgabe lauten:
`Die kleinste Zahl war die -21, die größte Zahl war die -21.`

Beachten Sie, dass die 0 bei der Suche nach der kleinsten und größten Zahl nicht berücksichtigt wird. Bei sofortiger Eingabe der 0 soll die Ausgabe „`Es wurden keine Zahlen eingegeben!`“ lauten.

Geben Sie Ihren Entwurf als Struktogramm an. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich wäre, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: Eine Umsetzung in C/C++ ist hier *nicht* gefordert.

Polynome: In dieser Aufgabe ist eine Funktion `printpoly(.,.)` zu entwerfen und in C- oder C++-Code umzusetzen.

Die Funktion soll dabei ein Array und dessen Größe als Parameter bekommen: Das Array enthält `int`-Werte, die die Koeffizienten eines Polynoms darstellen, wobei an Index i der Koeffizient für x^i steht. Die Funktion soll das Polynom auf dem Bildschirm ausgeben. Beispiele: zu dem Array `(4,1,0,-1,-2,3)` soll die Ausgabe „`f(x)=+3*x^5-2*x^4-x^3+x+4`“ lauten, zu dem Array `(0,0,2,0,-1)` soll die Ausgabe „`f(x)=-x^4+2*x^2`“ lauten.

Bei der Ausgabe sind folgende Regeln zu beachten (wie bei obigen Beispielen):

- ist der Koeffizient `1` oder `-1`, so soll der Faktor nicht angegeben werden, sondern direkt `+x^...` bzw. `-x^...`
- statt `x^1` ist nur `x` zu schreiben
- bei Koeffizient 0 wird der entsprechende Term weggelassen
- sind alle Koeffizienten gleich 0, so soll die Ausgabe „`f(x)=0`“ lauten

Hinweis: beachten Sie die Vorzeichen der Koeffizienten. Wie in obigem Beispiel ist für den ersten auszugebenden Koeffizienten ein führendes Pluszeichen erlaubt.

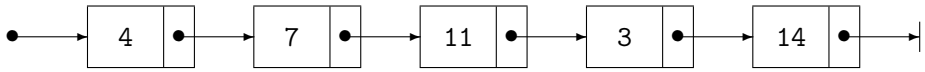
Geben Sie das Datenflussdiagramm (DF) der Ebene 0 an (Funktion als ein Kästchen), sowie einen Programmablaufplan mit Daten (PAD) oder ein Programmnetz (PN) für die von Ihnen entworfene Funktion mit den nötigen Verfeinerungen. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich ist, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: die Parameter sind an die Funktion zu übergeben und werden von dieser zurückgeliefert, Sie müssen sich *nicht* um die Eingabe der Werte kümmern. Geben Sie schließlich noch die Umsetzung der Funktion in C/C++ an.

Datenstruktur: In dieser Aufgabe ist eine **Datenstruktur** zu entwerfen, als Datenhierarchiediagramm darzustellen und in C/C++-Code umzusetzen. Die Datenstruktur soll zur Verwaltung der Bücher einer Bibliothek dienen und soll dabei die Daten für ein Buch beinhalten. Folgende Daten sollen gespeichert werden: Vor- und Nachname des Autors, Titel, Erscheinungsjahr, sowie das Datum, wann das Buch für die Bibliothek gekauft wurde (speichern Sie für das Datum jeweils Zahlen für den Tag, Monat und das Jahr). Das Datum soll dabei ein eigener Datentyp sein.

Geben Sie ein Datenhierarchiediagramm für Ihre Datenstruktur an und geben Sie die Umsetzung in C/C++-Code an.

Einfach verkettete Listen: In dieser Aufgabe beschäftigen wir uns mit einfach verketteten Listen, der Anker zeige dabei auf das erste Element.

1. Geben Sie eine C- oder C++-Typdefinition für eine einfach verkettete Liste aus `int`-Zahlen an. Der Datentyp für den Anker soll dabei `myList` heißen.
2. Schreiben Sie eine C/C++-Funktion `void dreierTest(myList liste)`, die als Parameter den Anker auf eine einfach verkettete Liste bekommt. Hat die Liste weniger als 3 Elemente, so soll die Meldung „`Liste ist zu kurz.`“ ausgegeben werden, ansonsten soll ermittelt werden, ob für jeweils drei direkt aufeinander folgende Elemente der Liste die Summe der ersten beiden Zahlen gleich der dritten Zahl ist, und wenn ja, diese Zahlen ausgegeben werden. Zu der Liste



soll die Ausgabe lauten: „`Die Summe von 4 und 7 ist 11. Die Summe von 11 und 3 ist 14.`“

Beachten Sie: die Parameter sind an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe der Werte kümmern.

Beachten Sie weiterhin: die verschiedenen 3-Tupel können sich – wie oben im Beispiel – überlappen.

Schaltjahr auf Rigel 7: Auch auf anderen Planeten gibt es Schaltjahre – auf Rigel 7 gelten folgende Regeln

- Regel 1: ist das Jahr ganzzahlig durch 7 teilbar, so ist es immer ein Schaltjahr
- Regel 2: von den verbleibenden Jahren sind die ganzzahlig durch 5 oder ganzzahlig durch 12 teilbaren Jahre Schaltjahre (Regel 2a), mit Ausnahme der ganzzahlig durch 60 teilbaren Jahre (Regel 2b)

Somit sind z. B. die Jahre 28 und 42 aufgrund der Regel 1 Schaltjahre, ebenso sind die Jahre 15 und 24 (Regel 2a) Schaltjahre. Das Jahr 120 ist hingegen kein Schaltjahr (Regel 2b). Die Regel 1 wird stets zuerst angewendet (so ist das Jahr 420 ein Schaltjahr trotz Regel 2b).

Vervollständigen Sie das Struktogramm, sodass die Ausgabe „Ja“ erfolgt, wenn das eingegebene Jahr ein Schaltjahr auf Rigel 7 ist, und „Nein“, wenn es kein Schaltjahr ist. Sie dürfen die Anweisung zur entsprechenden Ausgabe mit „`"Ja"`“ bzw. „`"Nein"`“ abkürzen.

Das Struktogramm soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich wäre, d. h., dass alle Fallunterscheidungen und Schleifen (wenn benötigt) auch im Entwurf als solche vorkommen müssen. Beachten Sie: Eine Umsetzung in C/C++ ist hier *nicht* gefordert.

```
Name    : SchaltjahrRigel7
Eingabe : Jahr (die zu untersuchende Jahreszahl)
Ausgabe : "Ja"/"Nein" (nur auf der Konsole)

Jahr - ganzzahliger Wert
```

```
Lese Jahreszahl Jahr ein
```

```
...
```

Gut gefüllt: In dieser Aufgabe ist eine Funktion `fuellEinsUm(.,.)` im Kontext eines Füllproblems zu entwerfen und in C- oder C++-Code umzusetzen.

Die Funktion soll dabei zwei Arrays als Parameter bekommen: Das eine Array enthält Werte, die die derzeitige Befüllung (in Liter) einer Reihe von Gefäßen angeben, ein zweites Array enthält Werte, die die Kapazitäten (in Liter) der betreffenden Gefäße angeben. Die Funktion soll zunächst das erste nicht-leere Gefäß in dem Array finden. Dieses soll – wenn möglich – nach folgenden Regeln in ein anderes Gefäß gefüllt werden, sodass

- der komplette Inhalt in das Zielgefäß passt und
- wenn es mehrere solche Zielgefäße gibt, dasjenige Gefäß gewählt wird, sodass der freie Platz in dem Zielgefäß nach dem Umfüllprozess möglichst gering ist;
- gibt es kein solches Gefäß (keines der Gefäße hat ausreichend Platz oder keines hätte nach dem Umfüllen weniger Restkapazität als das Ausgangsgefäß vor dem Umfüllen), so bleiben die Inhalte aller Gefäße unverändert.

Gegeben sind die C/C++-Definitionen:

```
#define N 7
typedef unsigned int Vol[N];
```

Ein Beispiel: seien zwei Arrays wie folgt definiert:

```
Vol inh = {0,0,2,7,4,5,1};
Vol kap = {6,5,7,8,9,9,6};
```

Das erste nicht-leere Gefäß ist das Gefäß mit Index 2 (das dritte im Array). Dieses kann in alle anderen Gefäße außer dem mit Index 3 (Inhalt 7, Kapazität 8) gefüllt werden. Es würde in das vorletzte Gefäß (mit Index 5) gefüllt werden, da hier nach dem Umfüllen nur noch Platz für weitere 2 Liter bleiben (bei allen anderen Gefäßen würden mindestens 3 weitere Liter Platz haben).

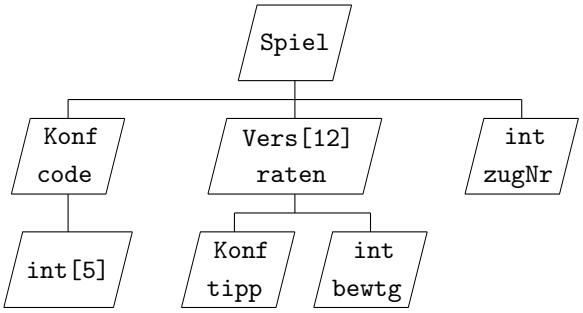
Beispiel 2: Hätte im obigen Beispiel das erste Gefäß die Kapazität 2 oder 3, so würde das Gefäß mit Index 2 in dieses erste Gefäß umgefüllt werden (kleinste Restkapazität von 0 bzw. 1).

Beispiel 3: Wäre der Inhalt des Gefäßes mit Index 2 groß gewesen, z. B. 7 Liter, so hätte es kein Gefäß gegeben, in das dieses komplett umgefüllt hätte werden können. Die Inhalte der Gefäße würden unverändert bleiben. Ebenso hätte man nicht umgefüllt, wenn der Inhalt 2 Liter und die Kapazität 3 Liter betragen hätte, da dann die Restkapazität (1 Liter) geringer wäre als die eines beliebigen anderen Zielgefäßes nach dem Umfüllen.

Der Aufruf `fuellEinsUm(inh,kap)` soll das Array `inh` so verändern, dass entweder das erste nicht-leere Gefäß nach obigen Regeln umgefüllt wird oder – wenn dies nicht möglich ist (oder es kein nicht-leeres Gefäß gibt) – die Ausgabe „`Umfuellen ist nicht moeglich!`“ erfolgen.

Geben Sie das Datenflussdiagramm (DF) der Ebene 0 an (Funktion als ein Kästchen), sowie einen Programmablaufplan für die von Ihnen entworfene Funktion mit den nötigen Verfeinerungen. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich ist, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: die Parameter sind an die Funktion zu übergeben und werden von dieser zurückgeliefert, Sie müssen sich *nicht* um die Ein- und Ausgabe der Werte kümmern. Geben Sie schließlich noch die Umsetzung der Funktion in C/C++ an.

Datenhierarchie und Modularisierung: Gegeben ist folgendes Datenhierarchiediagramm und eine Liste von Funktionen, die mit diesen Datentypen arbeiten. Ihre Aufgabe ist hier, die Headerdateien (Dateiendung `.h`) für die (nach der in Vorlesung und Labor vorgestellten Standardmethode) resultierenden Module anzugeben. Die Datenstrukturen sollen dabei vollständig in den Headerdateien implementiert werden. Ordnen Sie die Funktionen den entsprechenden Modulen zu und achten Sie darauf, welche Module welche anderen Module in der Headerdatei einbinden müssen. (Die Implementierung der Funktionen selbst ist hier nicht gefordert, auch können Sie hier ausnahmsweise auf Kommentarzeilen in den Headerdateien verzichten.)




```
void neuesSpiel(void);
void putSpiel(const Spiel & spiel);
void getBewtg(Vers & rat, const Konf code);
void getKonf(Konf & code);
void putKonf(const Konf code);
```

Die folgenden Hinweise sind nur relevant, wenn Sie die Aufgabe nicht komplett lösen können:

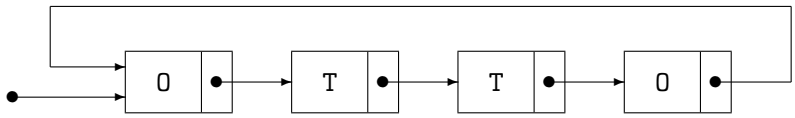
- Falls Sie die Aufteilung in einzelne Module nicht durchführen können, so geben Sie nur die Umsetzung des Datenhierarchiediagramms in C/C++-Code an.
- Falls Sie das Datenhierarchiediagramm nicht in C/C++-Code umsetzen können, so beschreiben Sie in 4–5 Sätzen, nach welchem Prinzip Sie auf Basis des Datenhierarchiediagramms eine Aufteilung in Module und die Zuordnung der Funktionen zu den Modulen

vornehmen würden. (Diese Erklärung ist nicht nötig, wenn Sie die obige Aufgabenstellung komplett lösen.)

Einfach verkettete Ringlisten: In dieser Aufgabe beschäftigen wir uns mit einfach verketteten Ringlisten, der Anker zeige dabei auf ein beliebiges Element der Ringliste.

1. Geben Sie eine C- oder C++-Typdefinition für eine einfach verkettete Ringliste aus Buchstaben an. Der Datentyp für den Anker soll dabei `charRgLstPtr` heißen.

2. Schreiben Sie eine C/C++-Funktion `void anz(charRgLstPtr liste)`, die als Parameter den Anker auf eine einfach verkettete Ringliste bekommt. Ist die Liste leer, so soll die Meldung „`Liste ist leer.`“ ausgegeben werden, ansonsten die Anzahl der Elemente in der Ringliste. Zu der Liste



soll die Ausgabe lauten: „`Die Ringliste enthaelt 4 Element(e).`“

Beachten Sie: der Parameter ist an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe der Elemente kümmern. Ihre Funktion braucht nicht zu überprüfen, ob es sich wirklich um eine Ringliste handelt.

Kleine und große Zahlen: Entwerfen Sie ein kurzes Programm, das von einem Benutzer Zahlen einliest und für diese die kleinste und größte Zahl bestimmt. Das Ende erfolgt durch eine Eingabe einer Zahl kleiner oder gleich 0 durch den Benutzer. Das Programm soll danach ausgeben, welches die kleinste und welches die größte vom Benutzer eingegebene Zahl war. Wurde die Eingabe sofort mit einer Zahl kleiner oder gleich 0 abgebrochen, so soll die Ausgabe lauten: `Es wurden keine Zahlen eingegeben.`

Beispiel: Der Benutzer gibt folgende Zahlen ein (jeweils getrennt durch die `<RETURN>`-Taste): `42 11 7 21 18 4 3 -2`, dann soll die Ausgabe lauten:
`Kleinste Zahl war die 3, größte Zahl war die 42.`

Geben Sie Ihren Entwurf als Struktogramm an. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich wäre, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: Eine Umsetzung in C/C++ ist hier *nicht* gefordert.

Datenstruktur: In dieser Aufgabe ist eine Datenstruktur zu entwerfen, als Datenhierarchiediagramm darzustellen und in C/C++-Code umzusetzen. Die Datenstruktur soll zur Verwaltung einer Datumsangabe auf Rigel 8 dienen. Ein Datum auf Rigel 8 setzt sich zusammen aus einer Jahreszahl, einer Monatsangabe (möglich sind hier `A`, `B` und `C`) und dem Tag im Monat (jeder Monat hat 12 Tage).

Geben Sie ein Datenhierarchiediagramm für Ihre Datenstruktur an und geben Sie die Umsetzung in C/C++-Code an.

Datum auf Rigel 8: In dieser Aufgabe ist eine Funktion `dauer(.,.)` zu entwerfen und in C- oder C++-Code umzusetzen.

Die Funktion soll dabei zwei Datumsangaben (aus obiger Aufgabe) als Parameter bekommen und die Differenz der beiden Datumsangaben berechnen. Die Ausgabe soll dabei getrennt nach Jahren, Monaten und Tagen erfolgen. Ist ein Wert 0, so soll dieser in der Ausgabe übersprungen werden.

Beispiel 1: Ist das frühere Datum `(2010,A,9)` und das spätere Datum `(2010,C,9)` so soll die Ausgabe lauten: `Die Dauer beträgt 2 Monate.`

Beispiel 2: Ist das frühere Datum `(2008,B,12)` und das spätere Datum `(2010,B,2)` so soll die Ausgabe lauten: `Die Dauer beträgt 1 Jahr, 2 Monate, 2 Tage.`

Beispiel 3: Ist das frühere Datum `(2010,B,12)` und das spätere Datum `(2010,A,2)` so soll die Ausgabe lauten: `Das zweite Datum liegt vor dem ersten!`

Beispiel 4: Ist das frühere Datum `(2010,A,9)` und das spätere Datum `(2010,A,9)` so soll die Ausgabe lauten: `Beide Datumsangaben sind identisch.`

Beispiel 5: Ist das frühere Datum `(2010,F,0)` und das spätere Datum `(2010,F,0)` so soll die Ausgabe lauten: `Mindestens eine der beiden Datumsangaben ist ungültig!`

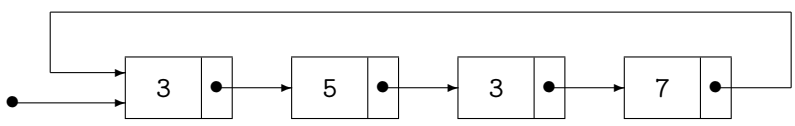
Geben Sie das Datenflussdiagramm (DF) der Ebene 0 an (Funktion als ein Kästchen), sowie einen Programmablaufplan für die von Ihnen entworfene Funktion mit den nötigen Verfeinerungen. Der Entwurf soll so detailliert sein, dass eine Umsetzung 1:1 in ein C/C++-Programm möglich ist, d. h., dass alle Fallunterscheidungen und Schleifen auch im Entwurf als solche vorkommen müssen. Beachten Sie: die Parameter sind an die Funktion zu übergeben und werden von dieser zurückgeliefert, Sie müssen sich *nicht* um die Ein- und Ausgabe der Werte kümmern. Geben Sie schließlich noch die Umsetzung der Funktion(en) in C/C++ an.

Hinweis: Falls Sie die vorherige Aufgabe nicht lösen konnten, verwenden Sie als Parameter für eine Datumsangabe entweder ein `int`-Array mit 3 Werten oder wahlweise drei einzelne `int`-Werte.

Einfach verkettete Ringlisten: In dieser Aufgabe beschäftigen wir uns mit einfach verketteten Ringlisten, der Anker zeige dabei auf ein beliebiges Element der Ringliste.

1. Geben Sie eine C- oder C++-Typdefinition für eine einfach verkettete Ringliste aus ganzen Zahlen an. Der Datentyp für den Anker soll dabei `RgLstPtr` heißen.

2. Schreiben Sie eine C/C++-Funktion `void anz(RgLstPtr liste)`, die als Parameter den Anker auf eine einfach verkettete Ringliste bekommt. Ist die Liste leer, so soll die Meldung „`Liste ist leer.`“ ausgegeben werden, ansonsten die Anzahl der verschiedenen Elemente in der Ringliste. Zu der Liste



soll die Ausgabe lauten: „`Die Ringliste enthaelt 3 verschiedene Element(e).`“

Beachten Sie: der Parameter ist an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe der Elemente kümmern. Ihre Funktion braucht nicht zu überprüfen, ob es sich wirklich um eine Ringliste handelt.

Hinweis 1: eine Möglichkeit der Lösung der Aufgabe ist, sich eine Hilfsfunktion zu schreiben, die überprüft, ob eine Zahl zwischen Anker und einem zweiten Element vorkommt. Überlegen Sie, wie Sie solch eine Hilfsfunktion zur Lösung der obigen Aufgabe nutzen können.

Hinweis 2: Sie können auch einen anderen Lösungsansatz verfolgen.