eXamen.press

Das C++ Kompendium

STL, Objektfabriken, Exceptions

Bearbeitet von
Gilbert Brands

1. Auflage 2010. Taschenbuch. xiv, 487 S. Paperback
ISBN 978 3 642 04786 2
Format (B x L): 15,5 x 23,5 cm
Gewicht: 1391 g

Kapitel 1
Zur professionellen Arbeitsweise

1.1 Arbeitsphilosophie und Methodik

Wir steigen zunächst etwas in eine „Arbeitsphilosophie“ ein, die den Hintergrund für die Ausführungen in den weiteren Kapiteln darstellt. Eine Sache gut und professionell zu betreiben bedeutet nicht, nur über ein gewisses Repertoire an Techniken zu verfügen, die blind nach Standardschema eingesetzt werden.[1] Man sollte nicht nur etwas anwenden können, sondern man sollte auch verstehen, in welcher Beziehung die angewandte Methode zum Problem passt und wo die Grenzen liegen.

1.1.1 Die Auswahl der Programmiersprache

Beginnen wir mit einigen Bemerkungen zur Sprachauswahl. Ihnen wird vermutlich schon aufgefallen sein, dass ich Sie sehr persönlich mit „Sie“ anrede, gemeinsame Gedanken mit „wir“ formulieren und mich bei Sachen, die einer gewissen Willkür unterliegen, der „ich“-Form bediene. Ich hoffe, Ihnen gefällt dieser Stil und Sie fühlen sich besser ermuntert, mir zu folgen (*oder zu widersprechen*) – aber das war eigentlich gar nicht mit „Sprachauswahl“ gemeint, sondern es geht um die verwendete Programmiersprache.

Jedes Programmiersprachenkonzept erlaubt natürlich eine besonders elegante Formulierung bestimmter Zusammenhänge, die bei Übertragung in eine andere Sprache etwas umständlicher ausfallen. Das sollte Sie aber nicht zu einer grundsätzlichen Wertung einer Programmiersprache verleiten. Leider ist gerade das häufig in Diskussionen zu beobachten, in denen an die Stelle aufgabenbezogener sachlicher Bewertungen, bei denen man durchaus zu dem Schluss kommen kann, dass ein

[1] Leider ist gerade das heute eine Erwartungshaltung und wird sogar vielfach unterstützt. Ein Student erklärte mir meine Aufgabe als Mathematik-Dozent einmal so: „Sie schreiben Formeln an die Tafel. Dann geben Sie uns Zahlen. Die setzen wir in die Formeln ein und rechnen etwas aus. Mehr brauchen wir über die Mathematik nicht zu wissen.“ Konsequenterweise erhält man später auf die Frage, wie ein bestimmtes Problem zu lösen ist, die vollständig korrekte Antwort „jemand fragen, der davon Ahnung hat.“

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_1, © Springer-Verlag Berlin Heidelberg 2010

bestimmtes Entwicklungssystem sehr viel geeigneter ist als andere, fundamentalreligiös anmutende Streitereien treten. Bei einer pragmatischen Vorgehensweise lässt sich durch Analyse der Fragen

- Wie ist die genaue Aufgabenstellung (*formuliert in der Sprache des Anwendungsentwicklers als genaues Abbild des Kundenwunsches, das heißt ohne durch spontane Realisierungsideen hervorgerufene Einschränkungen*).[2]
- Können die Randbedingungen der Aufgabe eingehalten werden?
- Welche Bibliotheken stehen zur Verfügung?
- Welche Einarbeitungszeit ist zu veranschlagen?
- Welche Projektzeit ist zu veranschlagen?
- Sind ähnliche Folgeprojekte zu erwarten (*die dann entsprechend schneller umgesetzt werden können, was einen hohen Einarbeitungsaufwand im ersten Projekt ausgleicht*).

meist recht schnell klären, ob eine bestimmte Programmierumgebung für die Lösung einer Aufgabe in Frage kommt. Ohne verbale Körperverletzung begehen zu müssen, stellt man dann fest, dass Java für die Programmierung von Fensteroberflächen gewisse Vorteile gegenüber FORTRAN besitzt, das wiederum mit großen partiellen Differentialgleichungssystemen besser zu Rande kommt als VisualBasic, und dass die Einarbeitung in Lisp vielleicht doch so lange dauert, dass C++ trotz längerer Entwicklungszeit das bessere Werkzeug ist.

1.1.2 Anforderungen an eine Anwendung

Sobald man die ersten Programmierschritte hinter sich gelassen hat und zu den ersten Aufgaben kommt, hinter denen eine echte Anwendung mit einem Auftraggeber steht und die sich nicht mehr durch Zusammenklicken einiger Bibliotheksfunktionen lösen lassen, ist die Zeit der in Anfängerkursen verbreiteten „*was–wollen–Sie–denn–es–läuft–doch*“– oder „*das–kann–man–doch–auch–einfach–so–machen*“–Philosophie[3] vorbei. Man stellt dann schnell fest, dass guter Code in folgenden Kategorien Punkte sammeln muss:

- **Effizienz.** Die Aufgabe will in einer bestimmten Zeit und mit bestimmten Ressourcen erledigt sein.
- **Wiederverwendbarkeit.** Bei Auftreten der gleiche (*Teil*)Aufgabe muss die vorhandene Lösung ohne Zeitverlust einsetzbar sein.

---

[2] Es kann länger dauern, dies festzustellen. Ein Kunde weiß niemals so genau, was er eigentlich will, beziehungsweise kann sich beim besten Willen nicht vorstellen, dass der Entwickler bei Benennung eines Themas die 25 notwendigen Details drumherum nicht kennt. Meist sind mehrere Anläufe notwendig, bis der Kunde das aus dem Lastenheft entwickelte Pflichtenheft abzeichnet, und wer sich nur mit dem Lastenheft zurückzieht und ohne weitere Diskussion mit der Arbeit loslegt, sollte mindestens einen guten Rechtsanwalt haben.

[3] Das ist die höfliche Variante der Version „auf meinem Atari habe ich das aber anders gemacht“, vorgetragen auf einem Seminar über die Bearbeitung von Differentialgleichungssystemen.

- **Typunabhängigkeit.** Bei Auftreten einer ähnlichen Aufgabe sollte vorhandener Code nach Austausch von Datentypen im Deklarationsteil möglichst ohne weitere Änderungen einsetzbar sein.
- **Korrektheit.** Die Anwendung soll (*natürlich*) mit allen korrekten Eingaben auch ein korrektes oder korrekt interpretierbares Ergebnis liefern.

  Das hört sich simpel an, ist es aber nicht, wie folgende Beispiele zeigen: Ein korrektes Ergebnis ist sicher die Antwort Vier auf die Frage „*Wie viel ist Zwei plus Zwei?*“, nicht korrekt ist die Antwort „*Die Zahl π hat den Wert 3,100*“, korrekt interpretierbar ist aber wiederum „*Die Zahl π hat den Wert 3,100, der Rechenfehler beträgt ±0,05 Einheiten*“.

  Bevor also die Frage nach einem korrekten Ergebnis beantwortet werden kann, muss zunächst definiert werden, wie ein korrektes Ergebnis überhaupt aussieht.
- **Robustheit.** Eine (*Teil-*)Anwendung soll bei falschen Eingaben weder korrekt aussehende (*unsinnige*) Ergebnisse liefern noch einfach einen Programmabbruch erzeugen, sondern den Fehler lokalisieren, um eine Korrektur zu ermöglichen.

  Hierbei denken Sie sicher zunächst an Beispiele wie die Eingabe von Buchstaben an Stellen, in denen Zahlen verlangt werden, also eine Robustheit zur Laufzeit. Das ist zwar korrekt aus der Sichtweise des Anwenders, aus der Sichtweise des Programmierers müssen wir die Robustheit wesentlich weiter fassen und verstehen darunter auch Techniken, Programmkode so zu strukturieren, dass die Möglichkeiten, erst zur Laufzeit sichtbar werdende Fehler einzubauen oder bereits vorhandenen korrekten Code falsch einzusetzen, minimiert werden. Mit anderen Worten: Ein Design ist um so robuster, je mehr der Übersetzer logisch falsch eingebaute Kodeteile nicht akzeptiert, das heißt die Sprachsyntax mit der Anwendungslogik verknüpft.

  Der eine oder andere von Ihnen mag vielleicht spontan auch an Ausnahmen (*exception*) denken. Das ist aber nur bedingt richtig, und wir werden später ein wesentlich differenzierteres Bild von Ausnahmen erzeugen.

Bei Betrachten dieser Liste stellt man schnell fest, dass jeder Punkt einer eigenständigen Untersuchung bedarf, einzeln optimierte Aspekte aber kein optimiertes Ganzes ergeben. Unter dem Aspekt *Robustheit* sind andere Sachen zu überprüfen als unter dem Aspekt *Korrektheit*, und eine Optimierung unter dem Gesichtspunkt *Typunabhängigkeit* kann Einbußen bei der *Effizienz* verursachen. Psychologisch ist man trotzdem meist gut beraten, die Aspekte zunächst gedanklich sauber zu trennen und zu untersuchen, um anschließend in der praktischen Auswertung einen brauchbaren Kompromiss zu finden.

1.1.3 Der Fehlerbegriff

Alles das impliziert aber auch, dass „Fehler“ im Laufe der Entwicklung einer Anwendung in den meisten Fällen nicht vollständig vermeidbar sind, wobei der Begriff „Fehler“ im erweiterten Sinn zu verstehen ist, das heißt nicht einfach nur der Rechner stehen bleibt, sondern irgendeine der vorgegebenen Randbedingungen (Geschwindigkeit*, Korrektheit,...*) nicht erfüllt wird. Wenn sich aber Fehler in

ein Programm einschleichen, dann sollen sie nicht erst dem Anwender auffallen, sondern im Zuständigkeitsbereich des Entwicklers zu Tage treten und beseitigt werden können.[4] Darüber hinaus sind auch Effekte zu berücksichtigen, die weder dem Entwickler noch dem Anwender auffallen, sondern „bösen Menschen“, die sich in fremde Systeme mit unterschiedlichen Motiven „einhacken“. Gerade so etwas kann beispielsweise recht aufwändig werden: Wenn A und B sich unterhalten, kann das damit anfangen, dass selbst A oder B oder beide nicht ehrlich sind, sondern betrügen, C nur lauscht oder als Vermittler zwischen A und B aktiv wird, sich als A oder B ausgibt, einfach frontal angreift oder einen der beiden unterwandert oder durch Bestechung auf seine Seite bringt.

Die vermutliche Unvermeidbarkeit von Fehlern [5] soll nicht als Aufforderung verstanden werden, schlampiges Arbeiten zu sanktionieren. Eine Reihe verschiedener Effekte kann auch bei sorgfältiger Arbeitsweise zu Fehlern führen. Dagegen kann man nur systematisches Testen setzen, allerdings, wie die Erfahrung lehrt, auch nicht mit 100%iger Erfolgsgarantie. Eine grobe Klassifizierung der Fehlerquellen und Abhilfemöglichkeiten beinhaltet:

- **Eingabefehler.** Auch bei sorgfältiger Programmierung lässt sich nicht ausschließen, dass bei der Eingabe des Programmkodes Fehler auftreten, beispielsweise normale Tippfehler oder spontane „Verbesserungen“ der ursprünglichen Anweisungssequenz. Die Fehler müssen bei der Übersetzung nicht unbedingt auffallen, sondern werden erst durch unerwartete Ergebnisse bei der Ausführung des Programms sichtbar, das heißt möglichst beim Test

  Eine Gegenmaßnahme ist das Testen kleiner Kodeteile mit Daten, deren Berechnungsergebnis bekannt ist. Erst bei fehlerfreiem Funktionieren darf dieser Kodeteil in andere Anwendungsteile aufgenommen werden. Der Sinn frühzeitigen Testens ist evident: Für begrenzte Berechnungen lassen sich im allgemeinen auch relativ leicht Testdaten mit bekanntem Ergebnis finden, und bei Verwendung ausgetesteter Methoden sollte ein auftretender Fehler auf den Programmteil zurückzuführen sein, an dem noch gearbeitet wird.[6]

---

[4] Vielleicht fällt Ihnen auf, dass ich mich wiederhole. Ich befinde mich da aber in guter Gesellschaft: In einem Handwerkerbuch findet man die Anweisung: „Der Leim muss 24 Stunden trocknen. Der Leim muss 24 Stunden trocknen.“. Das war kein Druckfehler, wie der Verfasser anschließend feststellte. Manches muss halt wiederholt werden, damit es in seiner vollen Bedeutung gewürdigt wird, wenn auch vielleicht etwas subtiler als in dem Handwerkerbuch.

[5] Mir ist klar, dass eine solche Feststellung schlecht mit gewissen Paradigmen vereinbar ist, zumal sie anscheinend die Möglichkeit nimmt, sich über bestimmte Produkte über ein gewisses Maß hinweg aufzuregen. Der empörte Leser möge aber erst einmal weiterlesen. Er wird feststellen, dass er seine ursprüngliche Empörung möglicherweise nicht mindern, sondern nur anders begründen muss.

[6] An dieser Stelle sei bereits eine Warnung angebracht: Für den Test dürfen nicht (nur) Gefälligkeitsdaten verwendet werden, sondern als Tester überzeuge man sich auch davon, dass der Algorithmus mit „hässlichen“ Daten ebenfalls die erwarteten Ergebnisse liefert. Das ist nicht selbstverständlich.

- **Optimierungseffekte.** Eine Anwendung soll nicht nur fehlerfrei ablaufen, sondern auch bestimmte Anforderungen bezüglich der bearbeiteten Datenmenge und des Datenumsatzes erfüllen. Nachdem ein Programmteil fertig gestellt ist, schließen sich daher häufig noch Optimierungsschritte an. Die teilweise gegensätzlichen Ziele „Optimieren“ und „Anwendung narrensicher machen“ gleichzeitig im Auge zu behalten ist jedoch für das menschliche Gehirn nicht ganz einfach, zumal wenn sehr viele Fakten zu berücksichtigen sind. Es ist dann nicht ganz auszuschließen, dass die Optimierung eines Kodeabschnittes zu Fehlern in einem anderen bislang fehlerfreien führt.

  Als Kontrollmaßnahme kann der nicht optimierte Code zur Eichung dienen: Kontrollläufe mit beiden Kodes und verschiedenen Testdaten müssen die gleichen Ergebnisse liefern. Stimmen die Ergebnisse nicht mit der Theorie, aber untereinander überein, so liegt ein grundsätzlicher Fehler vor, stimmen die Ergebnisse der beiden Kodes nicht überein, ist bei der Optimierung etwas daneben gegangen.

  Die grundsätzliche Arbeitsweise – Standardkode erzeugen und erst danach optimieren – sollte nur in Ausnahmefällen geändert werden. Nicht optimierter Code ist meist relativ schnell zu erstellen und liefert damit automatisch eine Referenz. Außerdem ist a priori häufig nicht abschätzbar, wie weit eine Optimierung gehen sollte. Möglicherweise erfüllt der nicht optimierte Code bereits die Anforderungen der Anwendung, und eine Optimierung vergrößert zwar den persönlichen Ruhm, aber nicht den Inhalt des Bankkontos. Auskunft liefert die jeweils fertiggestellte Version mit entsprechenden Testdaten, und anstelle aufwendiger weiterer Entwicklungsarbeit genügt ein Kommentar mit einem Aufriss der noch vorhandenen Verbesserungsideen für kritischere Folgeaufträge.
- **Entwicklungsprozess.** Der Entwicklungsprozess ist meist nicht so linear, wie die Theorie der Softwareentwicklung das gerne sehen möchte. Erst nach mehreren Anläufen richtig artikulierte oder verstandene Kundenwünsche, konzeptionelle Verbesserungen des Planungsteams und eigene Ideen der Entwickler führen zu Änderungen des Kodes während des Entwicklungsprozesses, wobei nicht auszuschließen ist, dass bereits geschlossene Lücken wieder geöffnet (*oder neue produziert*) werden. Der GAU ist das Hinausschießen der Anforderungen über die vom Design vorgesehenen Grenzen. Häufig wird trotzdem weitergemacht und die ursprünglich klare Systemstruktur verwässert – mit der Folge eines starken Anwachsens der Fehleranzahl. Begegnen lässt sich dem nur sehr schwer. Eigentlich kann nur versucht werden, die Planungsphase so lange wie möglich offen zu halten (*das verlangt die Disziplin, sich vom Rechner weg zu halten*), kleine Versuche nicht in die Entwicklung ausarten zu lassen und bei Erweiterungswünschen erst mal zu stöhnen – selbst wenn es nicht notwendig ist – um weitere Wünsche in Grenzen zu halten.
- **Systemstruktur.** Oft wird übersehen, dass Anwendungen nicht alleine auf einem Rechner ablaufen. Wir haben es mit einem hochkomplexen System zu tun, das aus vielen Komponenten unterschiedlicher und voneinander unabhängiger Entwickler besteht, die nicht selten aufeinander aufbauen. Eine veraltete

Version einer genutzten Bibliothek, und schon kann das Verhalten der eigenen Anwendung von dem erwarteten deutlich abweiche.

Abschließend sei noch kurz auf den Vorwurf eines Programmierfehlers seitens des Benutzers einer Anwendung eingegangen. Recht hat er mit dem Vorwurf, wenn beispielsweise

- das Programm auf einen Eingabefehler oder auf das Einlesen einer von einer anderen Anwendung bereitgestellten Datei sauer reagiert, denn was man nicht selbst gemacht hat, sollte man besser kontrollieren;
- eine vorsätzlich falsche Eingabe in einem Netzwerk mit unkooperativen Nutzern (ein so genannter Hack) die Anwendung aus dem Tritt bringt.

Ein Programm sollte schon robust gegen versehentliche Fehleingaben und, so weit möglich, gegen absichtliche Fehlbedienungen sein. Nicht (unbedingt) berechtigt sind allerdings Klagen, wenn

- das Programm mit normalerweise nur in Eigenregie erzeugten Daten, die durch unsachgemäßes Anfassen durch den Anwender verfälscht sind, erst einmal weiterarbeitet, oder
- Reaktionen auf eigentlich nicht vorgesehene Zustände wie Division durch Null oder Überschreiten eines Indexbereiches nicht oder als harter Programmabsturz erfolgen.

C/C++ Anwendungen werden in der Regel dann konstruiert, wenn es um Effizienz geht, und die Effizienz durch Prüfungen, die bei sorgfältiger Programmierung des Nutzers völlig unnötig sind, wieder zu beseitigen, ist Unfug. Wer das System nicht bedienen kann und solche Fehler macht, soll halt die Finger davon lassen.[7]

1.1.4 Prüfen und Testen

Wie aus der Liste hervorgeht, besteht die Entwicklung einer Anwendung eigentlich aus zwei getrennten Prozessen: Der Entwicklung des Systems und der Entwicklung des Testsystem.[8] Wir können uns in den weiteren Kapiteln des Buches mehr oder weniger nur auf den ersten Prozess konzentrieren, deshalb seien an dieser Stelle noch einige Worte zum Testen angebracht. Es ist sinnvoll, beide Prozesse

[7] Soviel auch zu Sprüchen wie „Exceptions dienen zur Behandlung nicht vorhergesehener Fehler." Genau betrachtet bestraft man mit solchen Prüfungen die ordentlichen Programmierer zugunsten der Blödels.

[8] Eigentlich ist das eine grobe Vereinfachung. Wer schon einmal mit Großprojekten Bekanntschaft gemacht hat, wird sich nicht nur über die recht großen Teams, sondern auch über die Vielzahl der involvierten Abteilungen gewundert haben. Die Beschäftigung mit diesem sehr interessanten Organisationsaufgaben müssen wir aber der Softwaretechnologie überlassen.

zu trennen, und industrielle Auftraggeber beauftragen häufig unterschiedliche Unternehmen mit Entwicklung und Test einer Anwendung. Beide Prozesse verlaufen parallel, das heißt mit der Entwicklung des Anwendungsdesigns muss der Designer des Testsystems bereits mit der Konstruktion der Testfälle beginnen, um auf jedem Teilstück des Projektes Testdaten bereitstellen zu können. Nur so lässt sich sicherstellen, dass Anwendungsteile nicht nur mit „Gefälligkeitsdaten“ geprüft werden. Zusammen mit dem Entwicklungsprozess der Anwendung entwickelt sich auch die Testumgebung, das heißt Änderungen im Anwendungsdesign müssen sich auch im Testdesign widerspiegeln. Im „großen Systemtest“, wenn die neue Anwendung an die Seite bereits laufender tritt, können auch Tests der alten Anwendungen wieder aufgerollt werden, um die Konsistenz des Gesamtsystems sicherzustellen.

An dieser Stelle sei auch ein Wort zu den so genannten Betaversionen erlaubt, die manchmal als Verschieben des ordentlichen Tests auf den Anwender verstanden wird. Diese Anschauungsweise ist nur teilweise korrekt, wenn wir die Bemerkungen über den Entwicklungsprozess berücksichtigen. Ein Entwickler entwirft eine Software, die bestimmte Datenmanipulationen vornimmt, der Anwender ist an der korrekten Bearbeitung seiner Daten interessiert – und dazwischen können Welten liegen. Betrachten wir als einfaches Beispiel ein Textverarbeitungsprogramm: Der Entwickler verwendet für seine Tests vielleicht Dokumente von 50-60 Seiten auf einer Maschine, die keine anderen Aufgaben hat – spätestens der dritte Anwender schreibt aber schon ein Buch mit 500 Seiten und erwartet, dass sein Rechner nebenbei auch noch eine Hand voll anderer Anwendungen gleichzeitig verwaltet. Hier kommt nun die Betaversion ins Spiel: Der Anwender weiß, dass möglicherweise nicht alles fehlerfrei läuft,[9] aber er hat Einfluss auf den weiteren Gang der Dinge, in dem er seine Testerfahrungen an den Entwickler meldet, und dieser bekommt ohne Ärger Informationen, wie er weiterem Ärger aus dem Weg gehen kann. Zum Schluss der Aktion liegt ein Produkt vor, das der Entwickler verkaufen kann und das genau das macht, was der Anwender erwartet (*oder was er zumindest kennt und keine Überraschungen beinhaltet*), so dass beide durch die Betaversion zu den Gewinnern zu rechnen sind.

Wie aus den wenigen Zeilen zu entnehmen ist, bietet allein der Testprozess genügend Stoff für ein eigenes Buch und ist sehr eng an die jeweilige Anwendung gebunden. Denken Sie beispielsweise an die testweise Inbetriebnahme einer neuen Anwendung im laufenden Betrieb einer Produktion:[10] Die neue Anwendung darf, da sie im Test läuft, durchaus noch Probleme aufweisen, und sei es nur im Zusammenspiel mit den anderen Komponenten. Es muss jedoch absolut sicher gestellt sein, dass der laufende Produktionsprozess nicht gestört wird oder irgendwelche Daten verändert oder zerstört werden.

Wir werden uns im weiteren auf Themen der Softwareentwicklung beschränken und ich lege es Ihnen ans Herz, Ihren Code im hier beschriebenen Sinn

[9] Hierunter sind nicht nur echte Fehler zu verstehen, sondern auch nicht erfüllte Erwartungen.

[10] Das muss nicht unbedingt eine Fabrik mit technischen Anlagen sein: Eine Bank produziert beispielsweise mit ihren Systemen Dienstleistungen.

schrittweise parallel zur Entwicklung zu testen. Die gängigen Testmittel sind, noch einmal zusammengefasst:

- Testdaten mit einem bekannten oder leicht zu überprüfenden Ergebnis. Dabei muss klar sein, dass es sich nicht (*nur*) um Gefälligkeitsdaten handelt, sondern auch Problemfälle berücksichtigt werden.
- Einsatz eines Debuggers zum schrittweisen Verfolgen des Programmablaufs. Das ist für einen Anfänger oft keine triviale Aufgabe, da bei längeren Programmen zunächst ein geeigneter Aufsetzpunkt für den Debugger gefunden werden muss.
- Implementation einer „Trace–Funktion“, das heißt der Ausgabe von Kontrollinformationen während der Programmausführung in eine Datei. Das Hilfsmittel ist recht beliebt und wird meist so gestaltet, dass der Modus durch einen Compiler-Schalter aktiviert werden kann, allerdings machen die fest installierten Trace–Anweisungen den Code undurchsichtiger (*und müssen natürlich auch an einer geeigneten Stelle implementiert sein*). Ein anderes Problem bei dieser Testmethode kann die entstehende Datenmenge sein. Bei ungeschicktem Platzieren der Trace-Anweisungen können etliche Megabyte an Daten zur Interpretation anfallen – und die Entwicklung eines Spezialprogramms zu deren Interpretation gehört sicher nicht zu den Traumaufgaben bei der Anwendungsentwicklung.

Wenn es darum geht, korrekt funktionierende Anwendungen zu produzieren, sollten alle Möglichkeiten genutzt werden, die das Erkennen von Fehlern bereits in der Erstellungsphase erleichtern. Eine relativ simple Möglichkeit ist die Einhaltung bestimmter einfacher Regeln bei der Dokumentation, der Definition von Schnittstellellen und der Verwendung von Standardsprachelementen. Wichtig ist auch die Nutzung von Möglichkeiten, die die verwendete Programmiersprache bietet. Jede Unstimmigkeit, die dem Übersetzer etwa bei der Typprüfung auffällt und vom Anwender beseitigt oder bestätigt werden muss, verringert das Risiko von Problemen nach der Auslieferung.

Ersatzweise wird häufig das Ausnahmemanagement (*exception*) für die Fehlerbehandlung angeboten, wobei jedoch übersehen wird, dass das Ausnahmemanagement erst zur Laufzeit funktioniert und Fehler unter Umständen nur mit bestimmten Datensätzen findet, möglicherweise auch erst beim Kunden. Wer das Ausnahmemanagement tatsächlich so verstehen möchte, also als Mittel zum Aufspüren von vom Entwickler nicht bedachter Situationen anstatt als Steuerungsinstrument zur kalkulierten Behandlung erkannter ungewöhnlicher Zustände, hat den eigentlichen Einsatzzweck nicht verstanden.[11]

[11] Etwas ketzerisch könnte man auch fragen, wieso die Entwickler der C++-Sprache eigentlich den Begriff *exception=Ausnahme* und nicht *error=Fehler* gewählt haben. Sind sie möglicherweise ihrer eigenen englischen Muttersprache nicht mächtig?

1.1.5 Der Einfluss der Theorie

Nicht alles lässt sich mit technischen oder systematischen Kenntnissen erledigen. Um Kreativität zu entwickeln oder bestimmte Garantien übernehmen zu können, ist auch Hintergrundwissen notwendig.

Eine immer wieder geübte Kritik am Ausbildungssystem ist die angebliche Theorielastigkeit. Gefordert wird mehr „Praxisbezug“, womit meist die jederzeitig unmittelbar erkennbare Relevanz einer theoretischen Betrachtung für ein praktisches Problem gemeint ist.[12] Ergebnis dieser Indoktrinierung ist dann bei Studierenden oft die Haltung „das brauche ich nicht“, sobald Grundlagen vermittelt werden sollen.

Nun soll sicher nicht gleich jeder ein begnadeter Theoretiker sein, aber es ist natürlich immer sinnvoll, wenn der Programmentwickler ein wenig von der Materie versteht, für die er eine Anwendung entwickeln soll. Im technischen Bereich bedeutet dies, dass physikalische oder elektrotechnische Grundkenntnisse sehr hilfreich sein können, und bei numerischen Berechnungen sollte der Entwickler ein Verständnis für die Mathematik mitbringen (*nicht nur, um den Auftraggeber zu verstehen, sondern auch, um die Qualität seiner Arbeit nachweisen zu können*). Überhaupt hat sehr vieles, was auf einem Rechner abläuft oder ablaufen soll, einen mathematischen Hintergrund (*auch die Relationen einer Datenbank sind mathematische Formulierungen*). Erwartet man also von einem Programm, dass es tatsächlich das macht, was erwartet wird, so sollte der Entwickler über solides mathematisches Hintergrundwissen verfügen. Lücken auf diesem Gebiet lassen sich oft schon beim ersten Blick auf den Quellkode durch merkwürdig gestaltete Schnittstellen oder Datenstrukturen bemerken.

Gefordert werden heute auch immer mehr , worunter Teamfähigkeit, Organisationstalent und Weiteres verstanden wird. Allerdings ist auch dazu festzuhalten, dass sich Schlüsselqualifikation in der Regel auf vorhandene Fachkompetenz beziehen und niemand bei seinem Kunden Pluspunkte dadurch sammeln können wird, dass er in einstündiger freier Rede mit philosophischem Hintergrund erläutert, warum er nicht in der Lage ist, die gestellte Aufgabe zu lösen.[13] Es gilt daher, zunächst eine Fachkompetenz zu entwickeln, auf der aufgesetzt werden kann. „Mehr Schlüsselkompetenz“ bedeutet daher auf keinen Fall „weniger Fachkompetenz“, und „Teamfähigkeit“ bedeutet nicht, eine Aufgabe durch permanentes Zusammenhocken zu lösen (*wobei letztendlich oft nur einer arbeitet*), sondern eine klare

---

[12] Die Forderung kommt nicht nur von Außenstehenden, sondern auch von Insidern. Auf die Frage, wie das denn bei den nicht gerade seltenen Fällen geschehen soll, zu denen man ohne eine gehörige Portion reine Theorie überhaupt nicht hingelangt, erhält man, wenn man Glück hat, keine Antwort. Möglich ist aber auch „das ist fertig, damit muss man sich nicht mehr beschäftigen“. Nun ja, offenbar hat man zuerst Atomkraftwerke und -Bomben gebaut und anschließend Einstein, Schrödinger und Kollegen gebeten, mal zu klären, was da vor sich geht.

[13] Genauso wenig wird natürlich der Theoretiker gebraucht, der ohne Beachtung der Kundenwünsche sein Modell umsetzt, weil es ihm vom theoretischen Standpunkt seines Faches als elegant erscheint.

Trennung in Einzelaufgaben und ein transparentes, strukturiertes und termingerechtes Arbeiten der einzelnen Teammitglieder, so dass Austausch und Vertretung möglich sind.

1.2 (Wieder-)Verwendbarkeit von Code

Die Programmiersprache C steht in dem Ruf, sehr viel Unfug zuzulassen und damit sauberes Programmieren zu verhindern. Die erste Aussage ist korrekt (*als vorzugsweise zur* Systemprogrammierung*vorgesehene Sprache muss C einige Sachen zulassen, vor denen andere Sprachen zurückschrecken*), die Schlussfolgerung des unsauberen Programmierens aber Unfug, denn wenn der Programmierer nicht weiß, wie er die vorhandenen Sprachmittel einzusetzen hat, liegt es mit Sicherheit nicht an der Sprache, dass dann auch nicht das gewünschte Ergebnis herauskommt.

Das Vorurteil bezüglich C wird dann gleich auch auf C++ übertragen, obwohl C++ eine eigene Sprache ist[14] und man schon einiges an Compilerschaltern bewegen muss, um den gleichen Unfug wie in C zu produzieren (*gleichwohl ist es natürlich auch hier möglich, wenn man es denn darauf anlegt, sich kräftig daneben zu benehmen*). Genau betrachtet ist sogar das Gegenteil der Fall: in Bezug auf Typsicherheit ist C++ eine der schärfsten Sprachen und ermöglicht deutlich kontrollierteres Programmieren als Java, da vieles, was andere Sprachen durchgehen lassen, vom Compiler gar nicht erst übersetzt wird.

Bei der Erstellung von Programmkode ist es ratsam (*und in Teamprojekten ein Muss*), einige Regeln einzuhalten, die bei korrekter Anwendung schon viele Probleme vermeiden helfen, und genau das wollen wir unter „sauberem Programmieren“ verstehen. Eigentlich sollten sie Bestandteil einer Einführungsveranstaltung in eine Programmiersprache sein. Aus irgendwelchen Gründen werden sie aber in der Praxis häufig nicht so konsequent vertieft, wie dies wünschenswert ist.[15] Zu bemerken ist dies an Schnittstellen, die an den falschen Stellen angelegt und nur unzureichend definiert sind, an Kodeteilen in Implementierungen, die dort nichts zu suchen haben, und so unzureichend aufgebauten Dokumentation, dass nach einigen Monaten der Ruhe in der Entwicklung sich die Frage stellt, ob ein kompletter Neubeginn der Programmierung nicht günstiger ist als der Versuch, den alten Code wieder zu verstehen.

Gerade das Thema „Dokumentation“ fällt immer wieder unangenehm in „freiem Code“ auf, da Dokumentation Zeit benötigt, die dann natürlich für das Programmieren fehlt. Wie die Bezeichnung „freier Code“ schon ausdrückt, verdienen

---

[14] Diese Aussage findet man schon bei B. Stroustrup, Die C++-Programmiersprache, Addison-Wesley, und er sollte es wissen, denn er hat sie entwickelt.

[15] Möglicherweise gehört das eine oder andere zu den „Selbstverständlichkeiten“, die in Büchern wegen scheinbarer Trivialität nur am Rande behandelt werden und von denen die Dozenten der Kurse erwarten, dass die Studenten selbst drauf kommen oder es in Büchern lesen – ein Teufelskreis.

die Autoren nichts an den von ihnen zur Verfügung gestellten Bibliotheken (*außer unsterblichem Ruhm*), und entsprechend knapp fällt die Dokumentation aus. Der professionelle kommerzielle Entwickler wird Nachlässigkeiten dieser Art aber schnell an seinem Einkommen bemerken. Ein Grund, von vornherein sorgfältig zu arbeiten. Vertiefen wir also die vernachlassigten Themen an dieser Stelle.

Ein Problem mit C/C++ Code soll hier aber auch nicht verschwiegen werden: die häufig zu beobachtende starke Spezialisierung auf bestimmte Systeme. Nahezu jede Anwendung kommt mit einem größeren Konfigurationspaket dahergelaufen, um den Code auf einer der unzähligen Linuxversionen oder anderen Betriebssystemen kompilieren zu können (*wobei die Konfigurationspakete obendrein auch noch wechseln können*). Oft steht ein größerer kraftraubender Kampf bevor, bis ein Paket nutzbar ist. Ob das wirklich notwendig ist und viel an Performanz oder anderen Vorteilen bringt, darf meiner Ansicht nach ruhig bezweifelt werden.

1.2.1 Bibliotheksmodule

1.2.1.1 Strukturen und Klassen

Bereits sehr früh lernt man bei der Programmierausbildung, dass Anwendungen nicht aus einem großen Klotz aneinandergereihter Befehle bestehen, sondern in kleinere Teile – Funktionen, Objekte – zerlegt werden. Dazu ist zunächst eine Analyse der Daten notwendig: Zusammen eine bearbeitbare Einheit bildende Daten, und zwar nur solche, werden zu einer Struktur vereinigt, zum Beispiel ein Vektor als ein Feld von Zahlen zusammen mit seiner Länge. Die Vereinigung kann formal logisch oder durch eine Datenstruktur erfolgen:

Logische Gliederung	Strukturdefinition

`int len;  // Länge von`
`double * zahlen; // „zahlen"`
	`struct Zahlenfeld {`
	`  int len;`
	`  double * zahlen;`
	`};//end struct`

Meist entscheidet sich bereits an dieser Stelle bei der Festlegung der Datenstrukturen, welche Aufgaben der Anwendung später übertragen werden können, ohne in eine Fehlerfalle nach der anderen zu tappen. Es bedarf oft eines guten Hintergrundwissens beim Entwickler (-> *Theorie*), um eine Ordnung der zusammen gehörenden Teile zu schaffen, und eines disziplinierten Verhaltens, die Ordnung umzusetzen anstatt schnell die nächste noch freie Variable zu verwenden. Doch schauen wir uns an, wie es weiter geht.

C–Strukturen als Vorstufen von C++–Klassen, die die Daten mit auf ihnen operierenden Funktionen vereinigen, sind gegenüber einzelnen Variablen vorzuziehen, sobald mehr als zwei bis drei Variable zu gruppieren oder mehr als ein bis zwei Objekte gleichzeitig zu verwalten sind. Die Gruppierung von Objekten lässt sich

fortsetzen, wobei verschiedene Arten von Strukturerweiterungen zu unterscheiden sind. Strukturerweiterungen können

- rekursiv erfolgen, das heißt die Datenfelder einer Struktur können wiederum Strukturen sein, oder
- spezialisierend sein, das heißt eine speziellere Struktur übernimmt die Datenfelder einer vorhandenen und fügt neue hinzu.

Welcher der Erweiterungstypen zum Einsatz kommt, ist anwendungsabhängig und nur aufgrund einer Analyse der Objekteigenschaften zu entscheiden. Beispielsweise sind Punkte Bestandteile eines Polygonzuges und Kreise Spezialisierungen einer Figur.

Bestandteile	Erweiterungen

`class Punkt {`
`  double x,y,z;`
`};//end class`
`class Polygon {`
`  int len;`
`  Punkt * p;`
`};//end class`
	`class Figur {...};`
	
	`class Kreis: public Figur`
	`  {...};`

Auch bei normalen Gegenständen sollte klar sein, dass die Arbeitsstelle einer Person vermutlich nichts in einer Struktur zu suchen hat, die sein Auto beschreibt und ein Radio Transistoren als Bestandteile enthält, sich aber nicht von ihnen ableitet.

Aufgabe. Entwerfen Sie ein Modell für eine Garage mit einem Auto darin. Versuchen Sie, das Fahrzeug und die Garage durch mehrere Kategorien zu klassifizieren.

1.2.1.2 Schnittstellendesign

Sind die Datenstrukturen festgelegt, so können Kodeteile zu Funktionen zusammengefasst werden, die sich durch folgende Eigenschaften ausweisen:

- Bearbeitet werden vollständige Datenstrukturen.
- Ausgeführt werden abgeschlossene Aufgaben.
- Die Aufgaben zeichnen sich durch wiederholtes Auftreten oder durch Zugehörigkeit zu einer bestimmten Aufgabeklasse aus.

  Beispielsweise können gleiche Aufgaben an beliebigen Stellen durch einen Funktionsaufruf durchgeführt werden, oder die Ein- und Ausgabe von Daten kann von Berechnungen getrennt werden.

Wesentlich ist die vollständige Bearbeitung einer abgeschlossenen Aufgabe innerhalb einer Funktion. Das Auftrag gebende Programm übergibt bestimmte Daten an die Funktion und erwartet die Rückgabe eines Ergebnisses. Werden beispielsweise die reellen Lösungen einer quadratische Gleichung gesucht und stellt die Bearbeitungsfunktion für die spezielle Gleichung

$$x^2 + 3x + 9 = 0$$

fest, dass die Lösung komplex ist, so nützt es wenig, wenn innerhalb der Funktion dies mit der Anweisung

```
printf("Die Lösung ist komplex\n");
```

bekannt gegeben wird, da der rufende Programmteil vermutlich nicht in der Lage ist, den Bildschirminhalt zu lesen. Die Aufgabe „*berechne die Wurzeln der quadratische Gleichung* " ist daher zu ergänzen durch „*und gebe die Art der Lösung an*". Eine sehr allgemein gehaltene Schnittstelle für eine C/C++ - Bibliothek bekommt so das Aussehen

```
enum solution {null_poly, const_poly, lin_poly,
               real1, real2, complex, gt2 };
solution solve_qe(Polynom<double>const& p,
                  double& r1, double& r2);
```

und die Implementation enthält keine Druckanweisungen. Die Eingabe ist hier nicht durch die Parameter der so genannten PQ–Formel, sondern als Polynom beliebigen Grades definiert, und der Leser verifiziere, dass alle verschiedenen Fälle erkannt und auch alle Lösungen, sofern sie existieren, ausgegeben werden.[16]

Ein anderes Beispiel ist die Erfassung eines Feldes, beispielsweise eines Vektors, wie er weiter oben definiert wurde, durch einen Dialog. Eine Dialogroutine muss natürlich die Größe des Feldes abfragen, das Feld erzeugen oder zumindest kontrollieren, dass alles hineinpasst, und anschließend alle Komponenten einlesen. Die meisten von Ihnen werden das Beispiel wohl für einen Witz halten, aber ich habe es schon mehrfach erlebt, dass ein Bearbeiter die Aufgabe auf mehrere Funktionen aufgeteilt hat, ohne dass von anderen irgendein Widerspruch zu hören war.

Wie bei Strukturen sind Schachtelungen von Funktionen möglich, als Spezialfall auch Schachtelungen der gleichen Funktion (*Rekursion*). Ein Beispiel ist die Auswertung von Ausdrücken wie

---

[16] Dieses „Nerven" mit mathematischen Inhalten wird sich nicht ändern. Versuchen Sie zu ergründen, was mit „real1" und „real2" wohl gemeint sein wird und wie sich die Ausgabe der reellen von den komplexen Lösungen unterscheidet. Im Vorgriff auf spätere Kapitel sei der eifrige Interpret, der gleich eine Implementation für das Problem erstellen möchte, gewarnt, einfach zur so genannten PQ-Formel zu greifen! Die gehört nämlich eindeutig in den Bereich des Unfugs, den man veranstalten kann, ohne dass die Programmiersprache etwas dafür kann.

```
Ausdruck:    Text(Parameter)      -mit-
Parameter:   Text    -oder-      Ausdruck

Beispiel:  Text(Inhalt)  -oder-  Text(Parameter(Inhalt))
```

Eine Funktion, die den Ausdruck in die Anteile innerhalb und außerhalb der Klammern zerlegt, kann natürlich wiederum auf den inneren Anteil angewandt werden, wenn dieser ebenfalls wieder Klammerausdrücke aufweist.

Aufgabe. Realisieren Sie eine solche Funktion.

Die bisherigen Ausführungen wirken auf Sie sicher recht abstrakt, lassen sich allgemeingültig aber kaum genauer beschreiben, denn wie eine Datenstruktur zusammengesetzt ist und welche Aufgaben wiedererkennbar oder wiederverwertbar sind, zeigt sich erst bei der Analyse eines konkreten Problems.

1.2.1.3 Modularisierung

Für die Strukturierung einer Anwendung halten wir fest, dass sich ein Programm in Funktionen zerlegen lässt und sich die einzelne Funktionen zu größeren Einheiten, Module genannt, zusammenfassen lassen. Technisch wird dies durch Zusammenfassen verschiedener Funktionen in einzelnen Dateien realisiert, was eine übersichtliche Verwaltung bei Entwicklung und Pflege erlaubt. In C/C++ werden diese Dateien durch bestimmte Dateierweiterungen gekennzeichnet.

```
*.h            Schnittstellendatei (header file)
*.c            C-Implementation
*.cpp,*.cxx    C++ - Implementation
```

Die Dateien mit der Endung „**.h**“ enthalten

- Funktionsköpfe (*Funktionsdefinitionen*) ohne Code,
- Konstantenvereinbarungen,
- Makros, also Symbole, Compileranweisungen oder kurze Kodestücke, für die eine spezielle Abkürzung vereinbart wird,
- `inline`–Funktion , das heißt komplette Funktionen mit Code, der vom Compiler an anderer Stelle direkt ohne Funktionsaufruf eingesetzt wird,
- Vorlagenklassen (*templates*) ebenfalls mit dem kompletten Code.

Der Code der nur in Form der Köpfe definierten Funktionen gehört in die Dateien mit der Endung „`.c`“ oder „`.cpp`“, in denen auch modulweite Variable deklariert oder interne Konstantenvereinbarungen definiert werden können.[17] Je nach

[17] Die Begriffe „Definition“ und „Deklaration“ werden manchmal etwas durcheinander verwendet, deshalb hier eine Erläuterung: Eine Definition ist eine Festlegung, wie etwas aussieht oder verwendet werden soll, eine Deklaration ist die Benennung eines nutzbaren Objektes. Typen oder Konstanten werden somit definiert, Funktionen definiert und implementiert, und Variablen deklariert.

Programmiersprache oder Programmierart können auch noch weitere Dateitypen auftreten oder die separaten Implementationsteile fehlen.

Module werden vom Compiler einzeln und für sich alleine übersetzt und später vom Linker zu einem kompletten Programm vereinigt. Die Modularisierung bieten den Vorteil, nach Änderungen nur an einem Modul nur dieses neu übersetzen zu müssen (*bei Bibliotheken fällt oft danach sogar der Linkschritt fort, da die benötigten Funktionen erst zur Laufzeit geladen werden*). Module können dadurch von verschiedenen Programmierern beigesteuert werden, die sich nur über die Schnittstellen einigen müssen, ihren Code und damit oft auch ihr Know-How aber nicht offen zulegen brauchen.

Das gilt jedoch nicht für Templates, die einen großen Teil dieses Buches ausmachen. Hierbei handelt es sich um Programmiermuster, die vom Compiler erst während des Übersetzungsvorgangs in normalen Programmkode übersetzt werden. Da sie dazu natürlich vorliegen müssen, lassen sie sich nicht in Module kapseln und so ihre Interna auch nicht geheim halten.

1.2.1.4 Gültigkeitsbereiche

Neben der Zusammenfassung von Funktionalitäten auf öffentlich (*mehr oder weniger*) bereitgestellten oder bekannten Daten haben Module außerdem die Aufgabe, private Daten vor der (*Manipulation durch die*) Außenwelt zu Kapseln. Hierbei sind nicht die als `private` oder `protected` deklarierten Attribute von Klassen gemeint. Eine in einem Modul zusammengefasste Funktionengruppe oder die in der Anwendung erzeugten Objekte einer im Modul definierten und implementierten Klasse teilen sich in vielen Fällen eine Reihe von Ressourcen, die den Mitgliedern der Gruppe, aber niemand anderem zugänglich sein sollen. Im Implementationsteil eines Moduls besteht die Möglichkeit, solche gemeinsamen Ressourcen zu definieren und zu verwalten.

Wir demonstrieren dies an einem Beispiel einer Klasse zur Zählung von Objekten. In der Header-Datei, die in anderen Programmteilen eingebunden wird, ist eine Basisklasse für die Zählung definiert.

```
/* Header-Datei: zaehler.h */
class ObjCount {
public:
    ObjCount();
    ~ObjCount();
    static int n_obj();
};//end class
```

Die Anzahl der gezählten Objekte wird mit der statischen, das heißt als Klassenmethode ohne vorherige Deklaration eines Objektes der Klasse aufrufbaren Funktion „`n_obj()`“ abgerufen. Die Implementationsdatei enthält eine statische Variable, mit der mit Hilfe der Konstruktoren und Destruktoren die Anzahl der lebenden (*existierenden*) Objekte gezählt wird (*das Schlüsselwort* `static` *sorgt*

in C++ dafür, dass die Gültigkeit von Variablennamen auf das Modul beschränkt bleibt und der Linker nicht ins Grübeln kommt, wenn ihm der Name einer Variablen in einem anderen Modul erneut begegnet.)

```
/* Implementations-Datei: zaehler.cpp */
#include    "zaehler.h"
static int cnt = 0;
ObjCount()  { cnt++; }
~ObjCount() { cnt--; }
int ObjCount::n_obj(){ return cnt; }
```

`ObjCount` kann in anderen Klassen als Vorgänger verwendet werden, zum Beispiel

```
class NewClass: private ObjCount { ... };//end class
```

Wird ein Objekt von `NewClass` erzeugt, so erhöht sich der Zähler im Modul um Eins, wird ein Objekt vernichtet, so erniedrigt sich der Zähler. Die Variable `cnt` ist aber nur innerhalb des Moduls „`zaehler.cpp`“ bekannt und kann von anderen Funktionen nicht verändert werden. Da Klassen im Prinzip auch nichts anderes als die Bindung bestimmter Ressourcen an Funktionen sind, können Sie sich die Äquivalenz zwischen Klassen und Modulen sicher schnell klar machen und dabei auch einmal sicherstellen, dass die Definition der Klasse `NewClass` mit einem privaten Vorgänger `ObjCount` korrekt ist. Es ist nicht notwendig, den Vererbungsteil `public` zu definieren!

Die Zählklasse ist in dieser Form allerdings in der Praxis wenig brauchbar, da alle Objekte erbender Klassen ohne Differenzierung gezählt werden. Bei praktischen Einsätzen solcher Zählungen wird man aber voraussichtlich gerade an Differenzierungen interessiert sein. Wir kommen an anderer Stelle auf solche Techniken zurück.

> **Aufgabe.** Entwickeln Sie eine Klasse, die dieses ermöglicht. Als Hinweis sei eine Substitution des Konstruktors durch
>
> ```
> ObjCount()   -->   ObjCount(int k)
> ```
>
> gegeben.

Eine wichtige Ergänzung ist noch bezüglich des Innenlebens von „`zaehler.h`“ notwendig: Erben mehrere Klassen von `ObjCount`, so ist ein rekursiver Aufruf der Schnittstellendatei nicht auszuschließen mit der Konsequenz der Verwirrung des Übersetzers, wenn dies tatsächlich geschieht. Er würde dann die Definition von `ObjCount` mehrfach einlesen und mit der Fehlermeldung quittieren, dass ein bereits definiertes Symbol erneut definiert wird. Durch Definition von Makro-Marken ist dies zu unterbinden:

```
#ifndef __OBJ_COUNTER_MARKE__
#define __OBJ_COUNTER_MARKE__
```

```
...
...  // Klassendefinition von  ObjCount
...
#endif
```

Prüfen Sie bitte nach, dass die Schnittstellendefinitionen der Klasse `ObjCount` in der Dateihierarchie

```
c1.h:        #include "zaehler.h"
c2.h:        #include "zaehler.h"
main.h:           #include "c1.h"
                  #include "c2.h"
```

jetzt nur einmalig eingelesen werden, da der Übersetzer beim zweiten Durchlauf die Makro–Marke bereits kennt und den Teil zwischen `#ifndef` und `#endif` überliest. Voraussetzung ist natürlich, dass die Bezeichnung des Makros eindeutig ist und nicht zufällig auch in einem anderen Modul verwendet wird (*also möglichst komplizierte Namen verwenden*).

Anmerkung. In C++ sollte die Verwendung von Makros – im Gegensatz zu C – aber möglichst auf diesen Fall und systemspezifische Details beschränkt bleiben und für andere Fälle `template`-Klassen verwendet werden. Wir werden dies noch genauer beleuchten.

1.2.2 Dokumentation von Code

1.2.2.1 Versionsnummern

Neben der geschickten Aufteilung größerer Anwendungen in Funktionen, Module und Bibliotheken hängt die Verwendbarkeit von Code sehr stark von der Dokumentation ab. Da Software ein lebendes Produkt ist, beginnt dies mit Versionsinformationen,[18] die zu Beginn einer Datei als Kommentar eingefügt werden und typischerweise folgenden Umfang aufweisen:

```
Titel:    Objektzählung
Autor:    Gilbert Brands
Version:  1.0.0
Datum:    2.1.2002 / 2.1.2002 / 2.1.2002
Datei:    zaehler.h
```

[18] Die ultimative fehlerfreie Software, die nie wieder verändert werden muss, existiert abgesehen von einzelnen Funktionen nicht. Die Aufgaben ändern sich, und Bibliotheken, die sich nicht mit ändern müssen, brauchen dies von grundlegenden Funktionen einmal abgesehen in der Regel deswegen nicht, weil sie nicht mehr zum Einsatz kommen.

Die Bedeutung der ersten beiden Einträge ist klar. Sie beantworten die Frage, ob man sich den Inhalt des Moduls genauer bezüglich der gesuchten Algorithmen anschauen soll und wem man unhöfliche Worte zukommen lassen kann, wenn etwas nicht funktioniert. Die folgende Versionsnummer (*und das dazugehörende Datumfeld*) besteht aus drei durch Punkte voneinander getrennten Nummern, die folgende Information tragen:

- 1.0.**0**: Die letzte Kennziffer (*und der letzte Datumeintrag*) dient der Fehlerrevision beziehungsweise der Optimierung bei unverändertem Funktionsumfang. Treten Probleme bei der Nutzung auf, so kann anhand der Ziffer und des Datum überprüft werden, ob verbesserte Versionen vorliegen, die möglicherweise die Probleme nicht mehr aufweisen. Um dies zu entscheiden sind weitere Informationen notwendig, auf die wir weiter unten zu sprechen kommen.
- 1.**0**.0: Die mittlere Kennziffer bezeichnet abwärts kompatible Bibliotheken mit erweitertem Funktionsumfang. Bei Veränderung wird die letzte Ziffer wieder zurückgesetzt. Nutzer der Bibliothek können bei Veränderung der Ziffer davon ausgehen, dass vorhandener Code weiterhin (*fehlerfrei*) funktioniert.
- **1**.0.0: Die erste Kennziffer (*die übrigen werden bei Änderung zurückgesetzt*) kennzeichnet grundlegende Funktionsänderungen der Bibliothek. Anwender müssen bei Einsatz der neuen Version davon ausgehen, dass ebenfalls Änderungen in ihren Programmen notwendig sind, um weiterhin (*fehlerfrei*) zu funktionieren.

Je nach Strategie sind Bedeutungsverschiebungen oder Erweiterungen möglich, beispielsweise die Verwendung gerade und ungerader Nummern zur Unterscheidung von experimentellen oder stabilisierten Versionen. Die Herausgabe neuer Versionen bedeutet im übrigen nicht, dass nun die alte in den Reißwolf wandert. Im Gegenteil ist von jeder abgeschlossenen Version eine Archivierung durchzuführen, so dass jederzeit für Prüfzwecke darauf zurückgegriffen werden kann. Stellen Sie sich beispielsweise eine Fehlermeldung aufgrund von Realdaten vor, die nicht durch Testdaten abgedeckt ist. Tritt dieser Fehler bereits in älteren Versionen auf oder erst ab einem bestimmten Zeitpunkt? Das lässt sich nur überprüfen, wenn Probeläufe mit älteren Versionen durchgeführt werden können. Neben der Fehlersuche hat das auch Auswirkungen auf die Produktverfügbarkeit. Beispielsweise kann eine ältere Version hilfsweise zum Einsatz kommen, bis der Fehler gefixt ist.

1.2.2.2 Verwaltung von Code

Die Sicherung der einzelnen Versionen kann natürlich in Form einzelner Archive erfolgen, sobald eine Version abgeschlossen ist, jedoch erweist sich das in der Praxis oft als zu grob. Unterschiedliche Kodedateien werden ihre Versionsnummern nicht parallel ändern und deshalb nach einiger Zeit mit einem schlecht durchschaubaren Nummernsalat aufwarten, und bei der Arbeit in Arbeitsgruppen sind unter

Umständen viele Personen an vielen Dateiänderungen beteiligt, bis die neue Version einen offiziellen Status erhält.

Um dem zu begegnen, werden verschiedene Werkzeuge eingesetzt, die man auch dem Hobbyentwickler durchaus empfehlen kann. Ein Versionsverfolgung wie SubVersion erlaubt eine jederzeitige Sicherung des aktuellen Standes, selbst fehlerhafter Zwischenstände. Bei der Gruppenarbeit bieten solche Werkzeuge die Möglichkeit, in Arbeit befindliche Dateien als blockiert zu markieren, so dass andere Programmierer nicht unbeabsichtigt Änderungen einspielen können bzw. wissen, dass sich dort etwas tut.

Anstelle von vielleicht 20 verschiedenen nummerierten Versionen (*die natürlich weiterhin nach den diskutierten Regeln verwaltet werden sollten*) sammelt man so in relativ kurzer Zeit hunderte von Schnappschüssen, wobei jeweils nur die tatsächliche geänderten Dateien gesichert werden und von ihnen wiederum auch nur die Änderungen, so dass die verschiedenen Versionen sehr wenig Raum beanspruchen. Der Vorteil ist, jederzeit auf eine bestimmte Version zurücksetzen zu können, wobei das Verwaltungssystem alle Dateien zu dem gewünschten Zeitpunkt wiederherstellt. Da die Versionsverfolgungssysteme wie SubVersion allerdings immer linear arbeiten, d.h. die Versionsnummer eine einzelne fortlaufend größer werdende Zahl darstellt, und die Verzweigung in der Versionsnummerierung nicht mitmachen, müssen echte Verzweigungen im Versionsnummersystem durch das Führen unterschiedlicher Verzeichnisse im Verwaltungssystem realisiert werden.

In Arbeitsgruppen ist es oft nicht vermeidbar, dass verschiedene Bearbeiter die gleichen Programmdateien ändern. Das Versionsmanagement stellt solche Problemfälle fest, so dass nicht unkontrolliert eine Version durch eine andere in der Datenbank überschrieben wird. Spezielle Merge-Programme bieten dann die Möglichkeit, die Unterschiede zwischen den Dateien auf Zeilenbasis zu begutachten und den jeweils aktuellen Code in die Zieldatei zu übernehmen. Die Programme sind intelligent genug, auch in größeren Dateien neue, gelöschte oder verschobene Zeilenblöcke zu finden und zur Auswahl anzubieten und ungeänderte Blöcke zu übergehen.

> **Aufgabe.** Machen Sie einige Versuche mit SubVersion/Tortoise oder einem ähnlichen Programm und einem Merge-Programm wie WinMerge oder der Merge-Version in SubVersion. Vermutlich werden Sie bald bestätigen können, dass solche Werkzeuge selbst für den Privatprogrammierer recht interessant sind.

1.2.2.3 Fehlerdatenbank (bug report database)

Zur Entscheidung, ob eine neue Version einer Bibliothek eingesetzt werden soll, ist die Information notwendig, ob die beanstandete Funktion auch überarbeitet wurde. Dazu dient die Fehlerdatenbank, die Bestandteil der Moduldatei oder eine eigenständige Datenbank sein kann. Sie enthält beispielsweise die Einträge
Die Einträge können unterschiedlichen „Fehlerklassen“ zugeordnet werden, hier „bug“ für Fehler oder „opt” für Optimierungen. Mit anderen Worten und zum

Version	Datum	Name	Klasse	Text	Bearbeitet
1.0.0	10.02.02	Müller	bug	…	1.0.1, 12.02.02 Meier
1.0.1	15.02.02	Müller	opt	…	

wiederholten Mal: Nicht nur ein unbrauchbares Ergebnis ist als zu behebender Fehler zu betrachten, auch Code, der Zeitvorgaben nicht einhält, ist als fehlerhaft zu betrachten. „Bearbeitet" enthält die Versionsnummer der neuen Version, das Datum und den Bearbeiter.

1.2.2.4 Testdatenbank (test database)

Für komplexere Anwendungen sind häufig auch sehr komplexe Testszenarien notwendig, wie bereits festgestellt wurde. Anhand der Testszenarien kann entschieden werden, ob die Anwendung für einen bestimmten Einsatzzweck freigegeben werden darf.

Tests bestimmen aber auch den Ablauf des Entwicklungsprozesses. Einzelne Programmteile werden mindestens vom Entwickler, bei anspruchsvolleren Projekten auch bereits vom Tester geprüft. Nach einer Änderung oder Erweiterung des Programmes genügt es aber nicht, einen neuen Test zu machen! Da nicht selten buchstäblich mit dem Hintern das umgeworfen wird, was man zuvor mühsam mit den Händen aufgebaut hat, sind zusätzlich alle bereits gelaufenen Tests zu wiederholen. Da dies im Laufe der Zeit recht viele werden können, kostet das nicht nur Laufzeit, sondern auch Auswertungszeit, weshalb Standardisierung und Automatisierung auch des Testbetriebs notwendig ist.

Die Testinformationen können innerhalb der Module oder (besser) in eigenen Datenbanken geführt werden:

```
Testfall:        (Schlagwort)Beschreibung des
                 Testfalls
Testversion:     Version des Tests zur Kontrolle von
                 Testerweiterungen
Name:            Tester
Datum:           Datum des durchgeführten Tests
Programm:        Liste mit den Programmen des
                 Testfalls. Hier können auch weitere
                 Dokumentationen zum Testfall
                 vorhandenen sein
Daten:           Liste der eingesetzten Testdaten
Bibliotheken:    Tabelle der eingesetzten Bibliotheks-
                 funktionen und der Versionsnummern
                 der Bibliotheken.
Ergebnis:        Ergebnisbeschreibung, Daten usw.
```

Durch Abgleich mit der Fehlerdatenbank kann entschieden werden, ob ein neuer Test notwendig ist, wobei ein Testfall im Wiederholungsfall mit einigen Änderungen erneut in der Datenbank auftritt..

Das ist hier mit Test aber nur zum Teil gemeint: Vielfach müssen die funktionsmäßig getesteten Anwendungen in komplexeren Umgebungen mit anderen Anwendungen zusammenarbeiten. Die Kontrolle, ob der Entwickler denn nun wirklich verstanden hat, was der Anwender alles wollte, alle Einflussparameter bei der Integration auch korrekt erfasst sind, das System auch unter gewissen Stressbedingungen noch korrekt arbeitet und bei Fehlern – während der Tests oder später im Betrieb – immer ein konsistenter Systemzustand erreicht werden kann, ist ein weit größerer Aufwand, und diese Arbeiten sollte der Entwickler zumindest nicht mehr alleine durchführen.

1.2.2.5 Arbeitsliste (ToDo database)

Als weitere Informationsdatenbank kann eine Arbeitsliste hinzugefügt werden, in der geplante neue Funktionen und ihr voraussichtlicher Freigabetermin aufgeführt werden. Bei der Auflistung möglicher Details würden wir jedoch schnell in ein Zeit- und Arbeitsgruppenmanagement geraten, weshalb wir an diesem Punkt abbrechen und auf entsprechende Kapitel in Lehrbüchern über Softwaretechnologie verweisen.

1.2.2.6 Dokumentation

Neben diesem Verwaltungsrahmen, der die Kontrolle der Evolution einer Bibliothek erlaubt, ist die Dokumentation ausschlaggebend für die korrekte Verwendung. Zur Dokumentation ihrer Quellen aufgefordert machen viele Anfänger aus einer Seite Programmkode zwei bis vier Seiten dokumentierten Programmkode, der anschließend oft noch schlechter lesbar ist und immer noch kaum Hinweise enthält, was man mit ihm machen kann und wie er zu bedienen ist.[19] Andere Entwickler lassen den Code durch ein Werkzeug aufarbeiten (z.B. Doxygen), das Klassenhierarchien, Methodenlisten und über normierte Kommentare auch ein wenig über die vom Entwickler hineingebrachte Programmlogik enthält (falls er sich die Mühe gemacht hat, die Kommentierung im vorgesehenen Sinn vorzunegmen). Eine Dokumentation in unserem Sinne sollte jedoch noch etwas mehr enthalten.

Eine Dokumentation besteht in der Hauptsache aus zwei Teilen. Im ersten Teil wird die zum Modul gehörende allgemeine Theorie dargestellt, die sich in Form mathematischer Ausdrücke, logischer Beziehungen (*was im Grunde auch nichts anderes ist*) oder einfachen Ablauflisten darstellen lässt. Sie kann als größerer Kommentarblock zu Beginn der Header-Datei eines Moduls oder, schöner formatiert, aber dann bitte mit Referenzen in den Code, als separates Dokument bereitgestellt

[19] Andere Programmierer, darunter leider auch viele, die freie Systemsoftware schreiben, verzichten (*aus diesem Grunde?*) vollständig auf eine Funktionsdokumentation und verlesen dem Nutzer vor einer Seite Code eine weitere Seite Urheberrechte. Der Code bleibt dabei oft weiterhin recht nebulös.

werden.[20] Ist die Theorie in dieser Form geklärt, erübrigen sich fast alle Kommentare in den Quellen, da diese im allgemeinen nichts anderes sind als die Übersetzung der Theorie in die Programmiersprache unter Berücksichtigung einiger spezieller Regeln.

Die erste ist die Konstanz der Symbole. Ist in der Theorie das Skalarprodukt zweier Vektoren in der Form

$$s = \sum_{i=0}^{n} a_i * b_i$$

angegeben, so gibt es keinen Grund, den Code dazu so zu verfassen:

```
double vektor_a[100],vektor_b[100],prod;
int dim,z;
...
prod=0;
for(z=0;z<dim;++z)
      prod=prod+vektor_a[z]*vektor_b[z];
```

Die Symbole *s*,*i*,*a*,*b* sind auch im Code zu verwenden.

Die zweite Regel ist die ordentliche Strukturierung des Programms, bei der alle Anweisungen, die durch `{..}` geklammert sind, also jeder Anweisungsblock, um eine bestimmte Anzahl von Leerzeichen gegenüber dem Block, in dem der betrachtete liegt, eingerückt wird (*eigentlich gehört das bereits zur Programmiertechnik*):

So nicht	Aber so

`for(i=0;i<10;++i)`
`if(a[i]>10){`
`c=c*a[i];`
`}else{`
`d=d+a[i];}`
	`for(i=0;i<10;++i)`
	`  if(a[i]>10){`
	`  c=c*a[i];`
	`  }else{`
	`  d=d+a[i];`
	`  }`

Voraussetzung für die Dokumentation ist natürlich auch ein solides Verständnis des Entwicklers für die theoretischen Grundlagen, worauf bereits in der Einleitung hingewiesen wurde. Für ein Modul, das spezielle Funktionen und Konstanten für die Kontrolle von Rundungsfehlern definiert, könnte das so aussehen:

[20] Beispielsweise kann man in der externen Dokumentation nur wenige wesentliche Programmzeilen auflisten und zum detaillierten Studium eine Marke /*MARK_001*/ in der Dokumentation und an der betreffenden Kodestelle hinterlassen, so dass der Nutzer beide Dokumente parallel öffnen und studieren kann.

```
Konstanten/Hilfsfunktionen für Rundungskontrollen
=================================================
Bei Verwendung von Zahlentypen mit Rundungsoperationen
nach jedem Rechenschritt ist die Prüfung
      if(x == 0) ...                    (1)
eines Wertes auf Null zu ersetzen durch
      if(_abs(x)<epsilon*K) ...         (2)
"epsilon" ist die Genauigkeit der Zahlendarstellung, K ein
Faktor zur Anpassung an den ...
```

Mit der Theorie werden meist auch eine Reihe von Objekten oder Abläufen definiert, die sich in einer Liste entsprechender Funktionen oder Klassen im Modul wiederfinden und die an die Darstellung der Theorie angehängt werden kann. Die Liste erlaubt dem Nutzer eine schnelle Orientierung in der Bibliothek und kann daher auch schon erste Erläuterungen zu Funktionen enthalten.

```
Funktionen der Klasse KSpline
- spline_ok: Splinekoeffizienten sind für den
             Stützpunktsatz generiert
- points: Anzahl der Stützpunkte
- add_point: Stützpunkt an das Ende hinzufügen. Der
             Punkt muss gültig sein
- insert_point: Stützpunkt an der angegebenen
             Stellen einfügen
- erase_point: Stützpunkt entfernen. Alle
             Operationen machen den
             Koeffizientensatz ungültig
...
```

Oft sind Funktionskonstruktionen der folgenden Art notwendig:

```
class A {
    ...
public:
    void TuWas(){this->TuWasSpezielles();};
};//end class
class B: public A{
    ...
protected:
    void TuWasSpezielles(){...};
};//end class
```

`TuWasSpezielles()` macht die konkrete Arbeit, kann aber nicht direkt aufgerufen werden, sondern wird als virtuelle Funktion weit abwärts in einer Basisklasse bedient (*oder muss sogar in erbenden Klassen überschrieben werden*). Solche Zusammenhänge sind bei beiden Methoden darzustellen, ihr Zweck ist zu erläutern, um korrekte Erweiterungen zu ermöglichen.

Ein weiterer Dokumentationsteil folgt jeweils vor den einzelnen Funktionen oder Klassen im Schnittstellenteil und beschreibt die auszutauschenden Parameter sowie gegebenenfalls weitere für das Verständnis des Ergebnisses eines Aufrufs notwendige Informationen.

```
/*
Lösen einer quadratischen Gleichung mit reellen
Koeffizienten.
Koeffizientenübergange durch Übergabe eines Polynoms.
Rückgabewerte siehe "enum"-Definition. Parameter:
- rt1 enthält die NS des linearen Polynoms, rt2
  unverändert
- rt1 und rt2 enthalten die NS des reellen Polynoms
  (bei doppelten NS rt1=rt2)
- rt1 enthält den reellen Anteil der komplexen NS,
  rt2 den positiven imaginären Teil
- rt1 und rt2 unverändert in allen anderen Fällen.
*/
template <class T>
  rootQGL QuadGLReal(const Polynom<T>& pl,
                     T& rt1, T& rt2);
```

Eine weitere Systematisierung des Dokumentationsinhalts ist kaum möglich, es kann jedoch nur zu Ausführlichkeit geraten werden, insbesondere bei größeren Klassenbibliotheken mit tief geschachtelten Vererbungshierarchien.

Zur Unterstützung einer ausführlichen Dokumentation gibt es, wie eingangs erwähnt, eine Reihe von Softwarewerkzeugen, etwa dem der Programmiersprache JAVA angegliederten JavaDoc oder dem für C/C++ kostenlos verfügbaren Doxygen. Es genügt, in den Kommentaren bestimmte Steuerzeichen zu verwenden, um es diesen Werkzeugen zu ermöglichen, beispielsweise interaktive HTML-Dokumentationen, die zusätzlich Angaben enthalten, welche anderen Funktionen innerhalb einer Methode und wo die Funktion selbst genutzt wird, so dass sich der Nutzer ein ausführliches Bild über die bestehenden Abhängigkeiten machen kann.

> **Aufgabe.** Besorgen Sie sich Doxygen aus dem Internet. Dokumentieren Sie einige Ihrer bisherigen Beispielkodes und testen Sie die Nachvollziehbarkeit der Dokumentation. Das kann auch als Langzeittest durchgeführt werden: implementieren Sie irgendetwas Exotisches, dokumentieren Sie es und vergessen Sie es für ein halbes Jahr. Wenn Sie dann noch problemlos wieder in die Thematik zurückfinden, war Ihre Dokumentation in Ordnung.

1.2.2.7 Bezeichnungskonventionen

Im Quellkode sollte die Einhaltung bestimmter Schreibkonventionen für die unterschiedlichen Begriffe empfohlen werden. Verschiedene Bezeichnerklassen (*Variable, Funktionsnamen, Konstanten, Makros us w*) lassen sich leicht

durch ein einheitliches Schreibschema unterscheiden, zum Beispiel einheitliche Groß/Kleinschreibung an bestimmten Positionen usw.:

Bezeichnungsschema	Verwendung
`__SCHALTER__`	Compiler-Schalter in #define - Anweisungen
`MAXINT`	Konstantenvereinbarungen in #define-Anweisungen
`LeseDaten(..)`	Funktionsbezeichnungen
`max(a,b)`	Funktionsmakros
`i, j_1, mlen, nStart`	Integervariablen

Der merkwürdige Konjunktiv oben deutet aber schon auf Probleme hin, und tatsächlich ist die Befolgung dieses Rats nur schwer durchzuhalten, wenn unterschiedliche Bibliotheken zum Einsatz kommen. Variationen wie

```
einefunktion(...)
eine_funktion(...)
eineFunktion(...)
EineFunktion(...)
```

sind leider bei Wechsel der Bibliothek an der Tagesordnung, wenn auch die zweite Form zumindest in der STL eine gewisse Bevorzugung genießt, dann aber auch häufig ohne Unterscheidung zwischen Methoden und Attributen.

Variablen- und Funktionsnamen sollen sinnvoll und leicht verwendbar sein. Beispielsweise werden in der Mathematik Zählvariablen meist in den Buchstabenraum `{i,j,k,l,m,n,o}` gelegt, während reelle Größen oft mit `{a,b,x,y}` bezeichnet werden. Es verwirrt nur, wenn in Programmen `x` als ganzzahlige Zählvariable auftritt. Vielfach existiert ohnehin eine mathematische Beschreibung dessen, was berechnet werden soll, und nach unseren weiter oben angestellten Betrachtungen ist damit für die Variablenbezeichnungen bereits alles klar. Bei anderen Bezeichnungen ist etwas Nachdenken und Fantasie angebracht. Variablennamen wie

```
zeiger_auf_das_letzte_zeichen
```

bezeichnen zwar möglicherweise sehr genau, wozu sie dienen, haben aber den Nachteil, dass sie immer wieder in dieser komplexen Form eingegeben werden müssen und bei etwas aufwendigeren Programmzeilen wie zum Beispiel einer Divisionsformel für komplexe Zahlen dazu führen, dass die Anweisungen nicht mehr in eine Textzeile untergebracht werden können oder überhaupt unleserlich werden. Die Abkürzung `zadlz` ist jedoch vermutlich genauso einfallslos. Neben etwas Überlegung bei der sinnvollen Bezeichnung von Variablen ist in manchen Fällen eine Kommentierung bei der Variablendeklaration sinnvoll:

```
int lastSign ;   /* letztes Zeichen im String */
```[21]

Ein weiteres Normierungsmerkmal haben Sie sicher auch im Text bereits identifiziert: die Sprache. Auch wenn man kein Freund der Anglifizierung der deutschen Sprache ist, ist eine englische Bezeichnung oft gefälliger. Was nun benutzt wird, ist eigentlich nebensächlich, so lange ein einheitliches durchschaubares Schema dabei herauskommt.[22]

1.2.2.8 Strukturkonventionen

Auf die Strukturierung von Programmkode durch Einrücken sind wir schon eingegangen. Aus Gründen der Programmiersicherheit sei zusätzlich kompakter und vollständiger Code empfohlen, also zum Beispiel

```
if(a<b){
     r=r*1.1;
}//endif
for(i=0;i<10;i++){
     ...
}//endfor(i)
```

anstelle von { und } auf einzelnen Zeilen ohne Kommentar oder Weglassen des Klammerns bei nur einer Anweisung. Gerade letzteres führt nämlich oft zu Fehlern bei Erweiterungen. Fällt später auf, dass eine weitere Anweisung in der Schleife untergebracht werden muss, so ist

```
for(i=0;..)      -->      for(i=0;..)
    a=b;                       a=b;
c=d;                           b=e;
                             c=d;
```

zwar recht gefällig anzusehen, hat aber vermutlich nicht die gewünschte Wirkung. Sind die Klammern schon vorher vorhanden, passiert das weniger leicht.

Von den bei C-Programmierern sehr beliebten komplexen Verschachtelungen sollte auf jeden Fall Abstand genommen werden. Die Anweisung

```
a[--i]=a[i++]+(1<<i);
```

ist zwar rechts eindrucksvoll, wenn man Anfänger einschüchtern will, führt aber nicht selten zu unerwarteten Ergebnisse.

---

[21] Wer jetzt aber auf folgende Idee kommt, dem ist auch nicht mehr zu helfen: „*int i ; /* Integer-Zählvariable für for-Schleifen */*“. Im übrigen werden Sie vermutlich bereits bemerkt haben, dass ich die Anglizismen nur begrenzt einsetze und im Text so weit wie möglich die deutschen Begriffe verwende. Bei Variablen- und Typenbezeichnungen sind englische Begriffe aber meist unbestreitbar eleganter – vielleicht weil uns das Kauderwelsch dann nicht so auffällt.

[22] Unter der Anglizierung leidet nicht nur die deutsche Sprache, sondern die Verwendung als Universalsprache schadet im Gegenzug auch der englischen, wie Sprachwissenschaftler herausgefunden haben. Eine Fremdsprache beherrscht man eben nicht perfekt, was die Muttersprachler im Gegenzug auch häufiger zwingt, sich einfacherer sprachlicher Mittel zu bedienen.

1.3 Qualitätssicherung

Im Eingangskapitel haben wir die Systementwicklung und die Testumgebung als zwei parallel ablaufende sich ergänzende Prozesse diskutiert. In vielen Fällen mit komplexem theoretischen Hintergrund ist es sinnvoll, die Systementwicklung in zwei weitere Prozesse zu zerlegen: Die eigentliche Systementwicklung und die Qualitätssicherung als eine Art theoretischer Test.

Die Aufgabe der eigentlichen Systementwicklung ändert sich nicht: Theorie, Dokumentation und Programmkode sind zu entwickeln. Die Aufgabe der Qualitätssicherung ist die Überprüfung, ob die vorgelegte Theorie schlüssig (*widerspruchsfrei, fehlerfrei*) und der Code verständlich ist und genau das macht, was in der Theorie beschrieben ist. In gewisser Weise ähnelt dies einer Testphase, und der Qualitätssicherer soll sich im Normalfall ähnlich verhalten:

- Er vermerkt Inkonsistenzen, Fehler und Missverständlichkeiten, ist aber nicht für Korrekturen oder Lösungen zuständig.[23]

Die Arbeit des Qualitätssicherung hat allerdings nur bedingt etwas mit dem Testprozess zu tun: Im Test erfolgreicher Code kann bei der Qualitätssicherung auffallen, bei der Qualitätssicherung erfolgreicher Code muss nicht alle Tests bestehen (*weil zum Beispiel die theoretischen Annahmen nicht zutreffen*). Der Sinn ist, die Anwendung für die Zukunft wart- und wiederverwendbar zu machen, insbesondere auch unter dem Gesichtspunkt eines Teamwechsels. Als Verhaltensregel auf den Punkt gebracht:

- der Entwickler sollte sich bei seiner Arbeit vor Augen halten, dass der Qualitätssicherer nicht aufgefordert ist, darüber nachzudenken, was er sich bei Entwicklung gedacht haben könnte, sondern jeden Schritt sukzessive aus den vorhergehenden erkennen können muss.
- Als Qualitätssicherer halte man sich umgekehrt daran, auf sofortige Schlüssigkeit zu prüfen und nicht seitenweise ein Programm oder eine Dokumentation dahingehend zu untersuchen, ob das, was man findet, nicht doch korrekt laufen könnte.

Im Grunde ist die Qualitätssicherung damit gar nichts ungewöhnliches Neues, sondern nur eine Formalisierung dessen, was im Entwicklungsprozess ohnehin meist passiert: Die Diskussion mit anderen Teammitgliedern über die Aufgaben und Lösungen sowie deren klare Darstellung. Entsprechend kann man im Gegensatz zum eigentlichen Testprozess, bei dem Entwickler und Tester verschiedene Personen sind, nicht zwischen Entwickler und Qualitätssicherer unterscheiden. Entwickler A

[23] Natürlich beteiligt er sich in der Praxis mit Vorschlägen und Empfehlungen am Entwicklungsprozess, genauso wie in der Testphase die Beziehung zwischen Entwickler und Tester sich nicht auf ein reine JA/NEIN-Kommunikation beschränkt. Die Entscheidung liegt aber beim Entwickler, der letztendlich auch die Verantwortung trägt.

sichert die Qualität der Software von Entwickler B und umgekehrt, so dass beide einigermaßen sicher sind, das geliefert zu bekommen, was sie erwarten, und im Bedarfsfall auch für den Kollegen einspringen zu können. Das meiste von dem, was in diesen Eingangskapiteln als „Regel“ vorgestellt wird, lässt sich als Grundlage einer Qualitätssicherung verwenden. Bei Einhaltung kann die Dokumentation knapp gehalten werden und Diskussionen sind nicht notwendig. Alle davon abweichenden Vorgehensweisen sind zumindest ausführlich zu dokumentieren/begründen und mit dem Qualitätssicherer zu verhandeln.

Im Rahmen der Qualitätssicherung durchfallen sollte auch die bei einigen C-Programmieren verbreitete Marotte, möglichst viel in kryptischer Form in einer Zeile unterbringen zu wollen (*vergleiche Beispiel am Ende von Kap. 2.2*). Das wird bei einer Nachfrage, was damit bezweckt werden soll, meist als „persönlicher Programmierstil“ verkauft und ist meist ein Versuch, den Optimierer des Compilers zu überlisten.[24] Als Qualitätssicherer sollte man ruhig darauf bestehen, dass Zusammenhänge, deren Darstellung in der Theorie schon drei Zeilen benötigen, im Code nicht weniger Platz beanspruchen.

> **Aufgabe.** Hier hätten Sie vermutlich kaum eine Aufgabe erwartet, oder? Vermutlich ist es die Aufgabe, deren Bearbeitung den längsten Zeitraum in Anspruch nimmt, obwohl relativ wenig zu tun ist. Hier ist sie: Suchen Sie eine ältere von Ihnen bearbeitete Anwendung, die Sie Ihrer Ansicht nach sehr sorgfältig bearbeitet und dokumentiert haben. Prüfen Sie die Anwendung als Qualitätsprüfer im gerade besprochenen Sinn und verbessern Sie Code und Dokumentation. Legen Sie die Anwendung auf Wiedervorlage in ein paar Monaten und wiederholen Sie das Ganze, bis Sie als Qualitätssicherer nichts mehr auszusetzen haben. Führen Sie die Prozedur auch mit einigen der hier erarbeiteten Anwendungen durch (*wählen Sie dazu möglichst Kodeteile, die Sie nicht so oft benötigen. Bei oft benötigten Kodes ist zu viel Hintergrundwissen präsent*).

1.4 Schnittstellenkonventionen

In diesem Kapitel werden wir uns mit der Verwendung von Zeigervariablen und dem Datenaustausch zwischen Funktionen beschäftigen. Wir werden dabei einige formale Regeln definieren, bei deren Beachtung Fehler des Typs „*Speicherzugriffsfehler*“ oder „*Schutzverletzung*“ recht unwahrscheinlich werden.[25] Einige dieser Fehler

---

[24] Natürlich darf jeder die Mathematik oder Physik neu erfinden und alle Fragen nur noch in Quiaxilotl beantworten. Das Problem besteht nur darin, genug „Jünger“ zu finden, die das unterstützen. Wenn das nicht gelingt, wird derjenige entweder sehr einsam oder er bequemt sich doch zu dem zurück, was die anderen bevorzugen.

[25] Man kann natürlich auch darauf verzichten und versuchen, die Fehlermeldungen mit eingeworbenen Werbebannern wie „*Diese Schutzverletzung präsentiert Ihnen Suff-Bräu*“ Gewinn bringend zu vermarkten.

treten häufiger auf, wenn zunächst mit einzelnen großen Entwicklungsdateien, die alles enthalten, anstelle von Modulen gearbeitet wird.[26]

Beginnen wir mit der für Anfängerkurse typischen Arbeitsweise, zunächst alles in einer Datei abzuwickeln (*und möglichst noch globale Variable zu verwenden, was leider oft zu spät unterdrückt wird*). Die Frage, woher eine arme Funktion denn bestimmte Sachen wissen soll, die nicht in den Übergabeparametern oder lokalen Informationen festgelegt sind, wird oft mit „*das sieht man doch!* “ beantwortet. Irrtum: Nach Verteilung der Funktionen in verschiedene Module sieht eine Funktion meist nichts mehr von zuvor globalen Vereinbarungen, und bei der Verwendung mit Bibliotheksteilen anderer Entwickler fallen weitere implizite Unterstellungen über Umgebungsinformationen fort. Beispielsweise macht es mathematisch Sinn, bei der Deklaration

```
double * poly; int len;    // dies ist ein Polynom
```

die Variable `len` als Grad des Polynoms und nicht als reservierte Größe des Feldes zu betrachten. Ist das nicht dokumentiert und gehen die Meinungen der beteiligten Programmierer hierzu auseinander und werden als Folge davon dann Einträge in `poly` oberhalb des höchsten Koeffizienten in einem Teil der Anwendung nicht auf Null gesetzt, im anderen aber benutzt, so kann es durchaus zu komischen Rechenergebnissen kommen.

Wir werden im folgenden einige formale Regeln entwickeln, die eine eindeutige Interpretation und damit eine korrekte Verarbeitung erlauben. Bei Einhaltung der Regeln entfällt die Notwendigkeit einer ausführlichen Dokumentation; lediglich Abweichungen sind detaillierter zu kommentieren. Ein Nebeneffekt der Regeln ist damit eine Reduktion des Dokumentationsaufwands. In den Beispielen werden wir C und C++ - Code teilweise mischen; lassen Sie sich sich davon aber nicht verwirren.

1.4.1 Erzeugung und Vernichtung von Zeigerbereichen

Zeigervariable sind in C und C++ die Voraussetzung für dynamisches Ver-halten von Anwendungen, wobei unter dynamischem Verhalten die Unkenntnis der genauen Datenmengen, Datenstrukturen und Methodenabläufe zum Zeitpunkt der Programmerstellung und die selbständige Einstellung der Anwendung auf die konkreten Erfordernisse verstanden wird. Als einfachen Fall können Sie sich einen Textstring vorstellen, dessen Länge nicht bekannt ist und der während des Programms dynamisch erzeugt werden muss.[27]

---

[26] Diese Regeln sind übrigens genau das, was bei der Ausbildung mit Java auf der Strecke bleibt und Java-Programmierern das Leben außerordentlich schwer machen, sollten sie einmal gezwungen sein, C-Code zu entwickeln.

[27] Das lässt sich nicht durch Großzügigkeit wegdiskutieren. Die Deklaration *char s[40000]* wird locker durch die Eingabe von „*Krieg und Frieden*“ in Form eines einzelnen Datenstrings ausgehebelt.

Im Folgenden verwenden wir meist die C-Systematic mit den Funktionen `malloc` und `free` für die Beschaffung und Freigabe von Zeigerbereichen. Die Überlegungen gelten natürlich genauso für die C++ Operatoren `new` und `delete`.

1.4.1.1 Zeiger und Fehler

Bei der Arbeit mit Zeigern kann es zu mehreren schwerwiegenden Fehlern kommen:

(a) Aufgrund der Dynamik kann Speicherplatz erst zur Laufzeit des Programms bereitgestellt werden, wenn die notwendigen Größen bekannt sind. Dies erfolgt mittels der Funktion `malloc` (*oder* new*bei C++*) durch Anforderung der Ressourcen vom Betriebssystem und Rückgabe der Ressourcen an das Betriebssystem mittels der Methoden `free` (*oder* delete) nach Ende der Nutzung.

Erfolgt die Rückgabe nicht und wird weiterer Speicher für andere Variable angefordert, kommt es bei intensiver Freispeichernutzung nach einiger Zeit zu einem Speicherüberlauf, zum Anwachsen der Auslagerungsdatei und zu einer starken Abnahme der Verarbeitungsgeschwindigkeit wegen häufiger Plattenzugriffe und schließlich zum Blockieren des Systems.

Da heutige Systeme nicht gerade zimperlich mit Ressourcen ausgestattet sind, treten die Auswirkungen erst nach einer geraumen Zeit unter Last auf, das heißt bei eingeschränkten Testläufen erst beim Kunden.

(b) Wesentlich direkter wirkt sich der Versuch aus, Speicherplatz mehrfach freizugeben, d.h. auf ein `free` (oder `delete`) erfolgt ein weiterer `free`-Aufruf für die gleiche Speicheradresse. Da das Betriebssystem aus Perfomanzgründen nichts kontrolliert und der Inhalt der Speicherstellen, sofern sie dem Programm zugeordnet bleiben, sich auch nicht verändert, wird eine abermalige Rückführung des Speichers in den Pool des Betriebssystems versucht, was in der Regel zu irgendwelchen unerlaubten Zuständen in den Speicherallozierungstabellen und damit zum Prozessabbruch führt.

Das Suchen nach solchen Fehlern ist oft noch nervenaufreibender als im Fall (a), da man ja nicht nach etwas ausgelassenem, sondern etwas versehentlich verdoppeltem sucht.

(c) Bei Zugriff auf den reservierter Speicherplatz werden die Bereichsgrenzen nicht eingehalten und davor oder dahinter liegender Speicherplatz überschrieben. Bei fest reservierten Variablen führt dies zum Überschreiben des Inhalts der daneben allozierten Variablen und damit zu unsinnigem Programmverhalten. Bei Zeigervariablen bringt das Betriebsystem aber auch Marken hinter den Grenzen für seine eigene Verwaltung an und kann dann bei der Rückgabe des Speichers nicht mehr korrekt reagieren. Besonders tückisch sind sogenannte Speicherlecks: Durch Überschreiben der Verwaltungsmarke eines Speicherbereiches von wenigen Bytes entsteht bei Betriebssystem der Eindruck, riesige Speicherbereiche reserviert zu haben. Neue Anforderungen werden dahinter gepackt. Der Effekt ist ähnlich dem in (a) beschriebenen, jedoch meist wesentlich stärker

und kann unter ungünstigen Umständen innerhalb von Minuten die komplette Festplatte durch eine Auslagerungsdatei belegen.

Beide Effekte sind recht tückisch, da der Übersetzer nichts bemerkt und die Anwendung zunächst auch normal startet. Haben sich solche Fehler eingeschlichen, ist das Auffinden und Beseitigen häufig recht aufwendig.

1.4.1.2 Geltungsbereiche

Variable, auch Zeigervariable, die auf einen vom Betriebssystem geborgten Speicher oder Datenbereiche anderer Variabler zeigen, besitzen in einem Programm einen beschränkten Geltungsbereich,[28] der sich auf eine Funktion oder, im Fall von C++, gegebenenfalls auch nur auf einen Block oder einen Teil davon beschränkt.

C	C++	Geltungsbereich
`int func(){`	`int func(){`	Variable „s" bekannt
`  char * s;`	`  ...`	Variable „j" bekannt
`  ...`	`  { int * j;`	„j" verliert
`}`	`  ...`	Gültigkeit
	`  }`	„s" verliert
	`  ...`	Gültigkeit
	`}`	

Wenn eine solche Variable ihren Gültigkeitsbereich verlässt, besteht keine Möglichkeit mehr, ihren Inhalt zu lokalisieren und darauf zuzugreifen. Der vom Betriebssystem ausgeborgte Speicher bemerkt solche Gültigkeitsgrenzen nicht und bliebt erhalten – Fehler (a).

1.4.1.3 Zeigerarten

Kommen wir nun wieder zurück auf das Eingangsthema. Bevor auf einer Zeigervariablen irgendwelche Typ- oder Datenmanipulationen vorgenommen werden, sollte sichergestellt sein, dass sie auch auf einen sinnvollen Speicherplatz zeigt. Das ist zwar recht einfach zu gewährleisten, wie wir im weiteren zeigen werden, bereitet jedoch offenbar vielen Leuten aus mir nicht ganz verständlichen Gründen Probleme und muss als eines der Hauptargumente für die „Unsauberkeit" von C herhalten. Das Grundprinzip ist sehr einfach: Eine Zeigervariable, der in einer Funktion ein

[28] Wenn von der Möglichkeit, globale Variable außerhalb der Funktionskörper zu vereinbaren, abgesehen wird. Außer in sehr speziellen Anwendungsfällen macht das aber aus verschiedenen Gründen meist wenig Sinn, so dass als erste Regel definiert werden kann: „*Globale Variable sind nur in Ausnahmefällen zulässig*."

Speicherbereich des Betriebssystems zugewiesen wird, muss diesen vor Ende ihres Gültigkeitsbereiches entweder wieder freigeben oder an eine Variable in einer anderen Funktion zur weiteren Verwaltung übertragen:

```
char * func() {
  char * s;
  ...
  s = (char*) malloc(len); // Anforderung
  ...                      |
  free(s); // Freigabe ...  |   return s; // Übertragung
}
```

Wann „zeigt“ nun eine Zeigervariable auf einen eigenen, das heißt vom Betriebssystem angeforderten Speicherbereich und wann auf einen Datenbereich einer anderen Variable? Bei größeren Funktionen mit vielen Programmzeilen kann man sich da ohne Einhaltung bestimmter Konventionen oder einer Analyse des Kodes nicht so ohne weiteres sicher sein, insbesondere, wenn die Funktion einer Weiterentwicklung unterliegt. Es ist daher notwendig, beim Entwurf einer Funktion einen vielleicht am Anfang überflüssig erscheinenden Sicherheitsaufwand zu betreiben, der aus

- Initialisierung der Variablen in typischer Weise
- Kommentierung
- Statuskontrolle

besteht. Eine Initialisierung besteht aus der Zuweisung eines „unmöglichen“ Zeigerwertes, die Statuskontrolle prüft das Vorliegen dieses unmöglichen Wertes. Der Kommentar übernimmt Sonderaufgaben der Typerklärung und wird so integraler Bestandteil einer Qualitätssicherung. Dabei muss nicht jeder Wert kommentiert werden; für eine Qualitätssicherung ist ausreichend, dass die Standardfälle global festgelegt und später nur noch Sonderfälle kommentiert werden.[29]

Regel.

```
char * s = 0; // Variablen mit Initialisierungwird
              // Speicher des Betriebssystems
              // zugewiesen ;
              // die Initialisierung erfolgt mit
              // dem „unmöglichen" Wert 0
int * khelp;  // Variablen ohne Initialisierung
```

[29] Diese Regel enthält die Standardfestlegung als Kommentar. Wird diese Vereinbarung als globale Vereinbarung festgelegt, so brauchen nur Sonderfälle kommentiert werden, zum Beispiel wenn „khelp“ in der beschriebenen Form verwendet, aber trotzdem durch „int * khelp = 0“ initialisiert wird.

```
                // dienen zum Verweis auf
                // Speicherbereiche, die
                // von anderen Variablen verwendet
                // werden.
                // Die Festlegung des
                // Verwendungstyps kann nicht
                // geändert werden
...
if (s != 0){  // Speicherfreigabe nur, wenn der
    free();    // Initialwert nicht mehr vorliegt,
    s=0;      // Wiederherstellen des
}//endif      // Initialzustands
```

Die Freigabeprüfung ist zweckmäßigerweise als Makro `_free(..)` zu definieren, um Schreibaufwand zu sparen und Fehlerquellen zu beseitigen. Im weiteren Verlauf des Kapitels kommen noch einige Makros hinzu.

> **Aufgabe.** Fügen Sie in Ihre bereits erstellte Header-Datei mit „allerhand brauchbaren Sachen“ (*Fehler: Referenz nicht gefunden*) das Makro `_free(..)`. Vergleichen Sie die Ausführung über Zeiger mit der Vorgehensweise bei der Arbeit mit Dateien in C und fügen Sie gleich auch das Makro `_fclose(..)` hinzu.

Per Regelwerk schließen wir auch die Übergabe der Verantwortung für einen angeforderten Speicherbereich an eine andere Variable innerhalb einer Methode und die Zeigerarithmetik auf Eigentümervariablen angeforderten Speichers aus Sicherheitsgründen aus. Nicht zulässig für die oben deklarierten Variablen sind folgende Anweisungen:

```
free(khelp);    // verboten, „khelp" besitzt keinen
                // eigenen Speicherbereich
s++;            // verboten, da die Startadresse des
                // freizugebenden Speichers verloren
                // geht
```

Die Verwendung dieser Regel erlaubt es, innerhalb einer Funktion die formale Verwendung von Zeigervariablen schnell zu prüfen. Das Schreiben des Programmkodes kann formalisiert werden, in dem nach Festlegen des Verwendungstyps einer Variablen Initialisierung und Freigabe an den Anfang und das Ende der Funktion geschrieben werden.[30] Anstelle einer Verwendung einer Variablen für mehrere, unter Umständen auch noch unterschiedliche Arbeitstypen sind ausreichend viele eindeutige Variable zu deklarieren.

---

[30] Die korrekte Freigabe allen angeforderten Speicherplatzes kann bei vorzeitigem Arbeitsende durch ein „goto“ an den Anfang des formalistisch eingeführten „Aufräumbereiches“ sichergestellt werden, anstatt die Funktion sofort zu verlassen und hierbei ein paar Aufräumaktionen zu vergessen.

1.4.2 Typzuweisung (cast – Operationen)

Zeigervariable können auf andere Variable gleichen oder auch eines anderen Typs „zeigen“. Im zweiten Fall ist in der Regel ein `type casting` notwendig, um den Übersetzer zu einer Akzeptanz der Anweisung zu überreden. Außerdem ist einige Vorsicht notwendig, um Fehler (c) zu vermeiden. Bei der Festlegung der beschreibbaren Feldlänge beim `type casting` ist eine Befragung des Betriebssystems recht sinnvoll:

```
int len;
double r1;
unsigned int * i = (unsigned int*) &r1;
len=sizeof(double)/sizeof(unsigned int);
```

In diesem Beispiel verweist die Zeigervariable `i` nicht auf einen eigenen Speicherbereich, sondern auf den der fest deklarierten Variable `r1`. Probleme mit nicht zurückgegebenem Speicher sind hier nicht zu erwarten.

> **Aufgabe.** Testen Sie diese Beispiel. Lassen sich so alle Bits der Fließkommavariablen auslesen? Falls nicht, wechseln Sie den Datentyp der Zeigers. Stellen Sie fest, welche Bits für Mantisse, Exponent und Vor-zeichen verwendet werden.

Allerdings muss man beim `type casting` genau wissen, was man macht. Der Compiler schluckt beispielsweise meist ohne Meckern das folgende wenig sinnvolle Programm:

```
const char * cc = „Hallo";
char * c;
...
c=(char*)cc;
c[1]='0';
```

Erst beim Versuch der Ausführung der letzten Anweisung kommt es zu einem Laufzeitfehler (`Schutzverletzung`), und das kann je nach Gesamtkonfiguration erst zu einem unangenehmen Zeitpunkt beim Kunden geschehen. Das Compilerverhalten ist allerdings korrekt: Eine `cast`-Anweisung entbindet den Compiler von seinen Kontrollpflichten und überträgt die Verantwortung für die Folgen auf den Programmierer, und wenn der nicht genau weiß, was er hier macht ...

Dieser relativ freizügige Umgang mit Datentypen ist einer der Gründe für den Ruf von C, „sauberes“ Programmieren nicht zu unterstützen. Andere Programmiersprachen lassen das zwar grundsätzlich auch zu, führen aber gegebenenfalls zur Laufzeit noch einige Prüfungen aus, die Unfug zumindest erschweren. Hier wäre natürlich etwas mehr Kontrolle des Anwenders durch den Compiler insbesondere während der Entwicklung wünschenswert, und hier kommen dann einige wesentlich Eigenschaften von C++ zum Zuge. Zunächst einmal kontrolliert C++ die Wertzuweisungen von Variablen untereinander wesentlich pingeliger als C und

verlangt bei vielem eine Quittung durch ein `type cast`, dass man das auch tatsächlich so gemeint hat. Damit ist aber noch nicht Schluss: Über die einfachen und unkontrollierten `cast`-Konventionen aus C hinaus bietet C++ Mechanismen an, die mehr Kontrolle erlauben. Dazu werden spezielle systemdefinierte Vorlagen verwendet (*grundsätzliches zu Vorlagen folgt in Kap. 4. Die Typumwandlungsvorlagen sehen zwar aus wie gewöhnliche Vorlagen, sind jedoch Bestandteil des Sprachumfangs*). Für die Typkonvertierung sind vier Konvertierungsklassen durch verschiedene Operatoren definiert.

1.4.2.1 const_cast<..>

Für die oben beschriebene Umwandlung von `const char*` in `char*` existiert die Anweisung

```
c = const_cast<char*>(cc)
```

Konvertierungen dieses Typs sind in manchen Funktionsaufrufen notwendig, in denen klar ist, dass der Inhalt des übergebenen Parameters nicht verändert wird, aus bestimmten internen Gründen der Parameter aber nicht als `const` deklariert werden kann. Die `const_cast`-Anweisung besitzt eine Reihe von ihr vorbehaltenen Rechten:

- von den speziellen `cast`-Anweisungen darf nur sie `const typ*` in `typ*` umwandeln,
- sie wandelt nur `const typ*` in `typ*` um, das heißt `const typ1*->typ2*` oder `const typ->typ` wird vom Compiler nicht zugelassen.

Bei Verwendung dieses `cast`-Operators im oben angegebenen Beispiel können wir nun schließen, dass der produzierte Laufzeitfehler, der natürlich auch hierbei auftritt, kein Versehen sondern Absicht war, und den Programmierer nun um so heftiger beschimpfen.

1.4.2.2 reinterpret_cast<..>

Für die mit dem `const_cast` nicht mögliche Umwandlung verschiedener Zeigertypen ineinander ist ein weiterer Operator vorgesehen:

```
char * c; int * i;
...
i=reinterpret_cast<int*>(c)
```

Auch für diesen `cast`-Operator gelten spezielle Regeln:

- das Mischen mit `const`-Typen ist nicht zulässig (*es ist aber ein Mischen mit* `const_cast` *möglich:*

```
int * i; const char* c;
...
i=reinterpret_cast<int*>(const_cast<char*>(c))
```

Beide Umwandlungsoperatoren zusammen decken in etwa das Spektrum der C–`cast`- Anweisung ab, nämlich fast jede beliebige Umwandlung. Ob das, was dann weiter damit geschieht, noch irgendeinen Sinn macht, liegt natürlich in der Verantwortung des Programmierers. Zumindest das versehentliche Verwechseln von Konstantenspeicherplatz und Variablenspeicher ist aber nicht mehr möglich.

1.4.2.3 static_cast<..>

In objektorientierten Anwendungen geht es oft darum, Zeiger innerhalb einer Klassenfamilie ineinander zu überführen, und zwar in beide Vererbungsrichtungen. Eine weiterer Umwandlungsoperator sichert ab, dass die Klassen tatsächlich miteinander verwandt sind:

```
class A {...};
class B: public A {...};
...
A*a; B*b;
...
a = static_cast<A*>(b);
...
b = static_cast<B*>(a);
```

Den Sinn beider Konvertierungsrichtungen sollten Sie sich nochmals kurz überlegen. Die Konvertierung zur Basisklasse ist notwendig, wenn grundlegende Operationen durchgeführt werden sollen, die bereits vor der Definition der erbenden Klasse projektiert oder implementiert waren. Sind diese Operationen ausgeführt, ist möglicherweise eine Weiterarbeit auf der oberen Ebene notwendig, was die Aufwärtskonvertierung notwendig macht.

Der Typumwandlungsoperator stellt nur zur Übersetzungszeit sicher, dass ein Verwandschaftsverhältnis existiert. Ob das Verhältnis sinnvoll definiert ist und sich hinter der zweiten Anweisung hinter dem Zeiger `a` tatsächlich eine Variable des Typs `b` verbirgt, ist Angelegenheit des Programmierers. Sicher einsetzen lässt sich die `cast` -Methode daher nur im „upcast“ einsetzen, das heißt in der Wandlung in einen Zeiger auf eine der vererbenden Klassen.

1.4.2.4 dynamic_cast<..>

Eine sichere Wandlung in beide Richtungen ist nur unter zwei Voraussetzungen möglich. Soll die Typumwandlung

```
b = dynamic_cast <B*>(a);
```

sicherstellen, dass **a** auf eine Variable des Typs **B** zeigt, so kann die Prüfung nicht mehr zur Übersetzungszeit vorgenommen werden, da meist nicht feststellbar ist, wo der Zeiger genau herkommt. `dynamic_cast` kann den `static_cast`–Anteil seiner Aufgabe während der Übersetzung erledigen, muss den Rest aber zur Laufzeit ausführen. Möglich ist so eine Prüfung zur Laufzeit natürlich nur, wenn die Klassen polymorph sind, das heißt virtuelle Methoden enthalten.[31] Ist dies nicht der Fall, wird bereits ein Übersetzerfehler generiert. Zur Laufzeit wird die tatsächliche Klassen-zugehörigkeit einer Variable überprüft:

Korrekte Zuweisung	Unkorrekte Zuweisung
`A * a;`	`A * a;`
`B * b;`	`B * b;`
`a = new B();`	`a = new A();`
`b = dynamic_cast<B*>(a);`	`b = dynamic_cast<B*>(a);`
`// b enthält Adresse`	`// b enthält 0`

Entspricht die Variable nicht dem Zieltyp, wie im rechten Beispiel, ist das Ergebnis der Zuweisung je nach Systemeinstellung ein Nullzeiger oder eine Ausnahme.

Aufgabe. Analysieren Sie folgende Anweisungen und geben Sie die zugehörenden `cast`-Operatoren in C++ an:

```
double r[10];
int i,j;
i=(int)r;
j=(int)r[0];
/* ---------- */
class A {};
class B: public A {};
A* fu() { return new B();}
...
B * b = fu();
```

Fassen wir zusammen: Abgesehen vom `dynamic_cast` ermöglichen die anderen drei `cast` -Operatoren das Gleiche wie die einfachen Anweisungen in C,

[31] Bei Anlegen einer Variablen wird normalerweise nur der Speicherplatz für die Attribute angelegt. Da diese Informationen unstrukturiert sind, ist eine Prüfung nicht möglich. Bei polymorphen Klassen wird zusätzlich aber auch eine Tabelle der virtuellen Methoden angelegt. Zur Laufzeit wird nicht auf eine im Programm verankerte Funktionsadresse verzweigt, sondern auf die in der Tabelle hinterlegte Methode. Die gleiche Anweisungssequenz kann somit bei mehrfachem Durchlauf und Austausch der Variablen völlig unterschiedliche Abläufe ergeben. Durch Vergleich der Methodentabelle der in der cast-Anweisung angegebenen Klasse mit der zu der Variablen gespeicherten Methodentabelle ist nun in der Tat sicher feststellbar, ob die gewünschte Wandlung durchführbar ist.

jedoch kann im Code nun ohne großes Suchen überprüft werden, welche Konvertierungsklasse angewendet wurde und ob dies der Theorie entspricht. Unter diesem Sicherheitsaspekt sollten Sie die C++ - Operatoren anwenden, sobald Sie einen C++ - Compiler verwenden, und nicht aus Bequemlichkeit wegen des etwas geringeren Schreibaufwands die einfache C–Notation verwenden.

1.4.3 Eigentumsrechte

Im letzten Kapitel haben wir bemerkt, dass reservierter Speicherplatz, der innerhalb der reservierenden Funktion nicht freigegeben wird, an das rufende Programm übertragen wird. Dieses kann den Zeigerbereich erneut an weitere Unterprogramm oder an seine eigene Oberfunktion weiterreichen. Um die Zuständigkeiten bei der Nutzung von Speicherbereich in mehreren Programmbereichen zu klären, sind die Fälle Import und Export bei Wechsel der Funktion zu betrachten. Wir beginnen mit einer Untersuchung des Exports.

1.4.3.1 Export von reserviertem Speicherplatz

Regel. Die von einer nicht weiter gekennzeichneten Funktion zurück gegebenen Zeiger verweisen auf reservierten Speicherplatz, der wie eigener reservierter Speicherplatz zu behandeln ist.

Hauptprogramm	exportierende Funktion

`char * t = 0 ;`
`...`
`t = func(); // malloc()`
`// durch`
`// func()`
`// ersetzt`
`...`
`if (t != 0){`
`free(t);`
`t=0;`
`}//endif`
	`char * func() {`
	`char * s = 0;`
	`...`
	`s = (char*)`
	`malloc(len);`
	`...`
	`return s;`
	`}//end function`

Die Regel hat Konsequenzen, wenn Daten aus einem Teilbereich kopiert werden sollen. Wir können zwei Möglichkeiten unterscheiden, die durch folgenden Code repräsentiert werden:

```
char * func(char * s){
   char * t=0;
   ...
   t = (char*) malloc(len);
   ...
```

```
   strcpy(t,&s[k]);
   ...
   return t;
}//end function
char * sub=0;
sub=func(s);
...
_free(sub);
```

Die Funktion erzeugt einen Teilstring aus einem vorhandenen String. Dazu ist ein neuer Speicherbereich zu reservieren, in den der Inhalt hinein kopiert wird. Im rufenden Programm wird eine Variable mit Initialiserung und Freigabe (*hier als Makro-Aufruf*) verwendet.

1.4.3.2 Export von Zeigern ohne eigenen Speicher

In dem betrachteten spezielle Fall ist aber auch eine andere Implementation denkbar, die nicht der Regel folgt:

```
char * func_ptr(char * s){
   ...
   return &s[k];
}//end function

char * sub;
sub=func_ptr(s);
```

Der Rückgabewert der Funktion sieht zunächst genauso aus wie im ersten Fall, jedoch wird kein neuer Speicherplatz reserviert, sondern ein Zeiger auf die Position im vorhandenen zurückgegeben. Im rufenden Programm ist konsequenterweise eine Variable ohne Initialiserung und ohne Freigabe zu verwenden. Weiterhin sind im Gegensatz zum ersten Modell alle Än-derungen, die am Funktionsrückgabewert vorgenommen werden, auch im Original zu beobachten.

Anwendungsfälle dieser Art sind durchaus sehr sinnvoll und treten nicht unbedingt selten auf. Die beiden Fälle sind aber strikt auseinander zu halten.

Regel. Funktionen mit Zeigerrückgabewerten ohne eigenen reservierten Speicher erhalten spezielle Namenskennungen, hier beispielsweise die Endung `_ptr` oder im weiteren alle Funktionen mit dem Beginn `find_` .

Eine Kennzeichnung durch besondere Namenskonventionen ist sicherer als die (*auf jeden Fall durchzuführende*) alleinige Kennzeichnung durch Kommentare, da die korrekte Verwendung in diesem Fall ohne Hinzuziehen der Header-Dateien überprüft werden kann. Sie können solche Na-menskonventionen auch in professionellen Bibliotheken finden, wobei bestimmte Namensgebungen allerdings häufig nur für Teilbereiche gelten.

1.4.3.3 Import von Zeigern

Als Gegenstück zum Export ist der Import zu betrachten. Wie leicht nachvollziehbar ist, kann eine Funktion nicht feststellen, ob eine übergebene Zeigervariable auf festen oder von Betriebssystem geborgten Speicher verweist, darf ihn also auf keinen Fall freigeben:

Methode 1	Methode 2	importierende Funktion

`char * s = 0;`
`...`
`s = (char*)`
`malloc(len);`
`...`
`func(s);`
`...`
	`char s[LEN] ;`
	`...`
	`func(s);`
	`...`
		`int func(char * s){`
		`  ...`
		`  ???`
		`}`

Regel. Eigentümer eines Speicherbereichs ist der Programmbereich, der den Speicher angefordert oder ihn von einem Eigentümer in Form eines Funktionsrückgabewertes geerbt hat. Ein angeforderter Speicherbereich darf ausschließlich vom Eigentümer freigegeben oder an den den Eigentümer rufenden Programmteil übertragen werden, der damit neuer Eigentümer wird. Eine „Veräußerung“ des Eigentumsrechtes an gerufene Funktionen ist nicht zulässig.

1.4.3.4 Veränderung der Speichergröße

Allerdings müssen wir auch hier eine Ausnahme zulassen, die wie bei den Importregeln durch eine besondere Namenskonvention angezeigt wird. Ihre Notwendigkeit lässt sich leicht begründen: In komplexeren Programmabläufen tritt mitunter der Fall auf, dass die ursprünglich angeforderte Speichergröße nicht mehr den Bedürfnissen des Programms entspricht, die dort gespeicherten Daten aber weiterhin benötigt werden. Um die Arbeit weiterführen zu können, wird in diesem Fall ein neuer, ausreichend großer Speicherplatz angefordert, der Inhalt des zu klein gewordenen Bereich wird umkopiert und der alte Bereich freigegeben.[32] Kann dieser Fall an verschiedenen Stellen im Programm auftreten, so ist die Definition und Implementation einer Funktion sinnvoll, die diese Operation durchführt.

Regel. Eine Funktion, in der eine Größenänderung des reservierten Speicherplatzes vorgenommen wird, muss den durch den Parameter übergebenen reservierten

[32] Das Betriebssystem ist häufig auch in der Lage, zu prüfen, ob der fehlende Bereich im Anschluß an den belegten noch frei ist und kann die Reservierung entsprechend zu erhöhen. In diesem Fall entfällt das Kopieren.

Speicherplatz freigeben und den neuen Speicherplatz als Rückgabewert abliefern. Die Parametervariable muss über eigenen reservierten Speicherplatz verfügen, der Rückgabewert wird auf der gleichen Variablen wieder abgelegt, so dass diese weiterhin für die Ver-waltung zuständig bleibt. Die Funktionen sind ebenfalls durch spezielle Namenskonventionen eindeutig zu kennzeichnen.

```
// Funktionskopf fuer „Resize"-Funktion
// ==================================
MyType * ResizeMyType(MyType * old, int oldLen, int newLen){
   MyType * neu;
   neu=(MyType*) malloc(newLen*sizeof(MyType));
   memcpy(neu,old,oldLen*sizeof(MyType));
   free(old);
   return neu;
};//end function
// Nutzung im rufenden Programmteil
// ==================================
MyType * ptr ;
...
ptr = ResizeMyType(ptr,len,neulen);
...
```

Zur Kennzeichnung als „Ausnahmefunktion“ beinhaltet der Funktionsname das Teilwort `Resize`. Charakteristisch für die Nutzung ist ein Aufruf, bei dem die Übergabevariable gleichzeitig den Rückgabewert aufnimmt. Der Rückgabewert darf nicht von einer anderen Variablen aufgenommen werden, da der ursprüngliche Zeiger meist zerstört ist (*Ausnahme: Falls eine Erweiterung des alten Bereichs möglich ist oder* `oldLen>=newLen` *gilt, kann er weiterhin gültig bleiben*). Eine etwas ausgefeiltere Version der `Resize`-Funktion berücksichtigt auch diese Fälle oder vergrößert den Zeigerbereich nur in größeren Blöcken, ohne dass dies von außen ersichtlich wäre, und dient so auch zur Entlastung des Hauptprogramms von häufigen Größenvergleichen. Der Leser möge sie zur Übung entwerfen.

> **Aufgabe.** Im oben dargestellten Beispiel wird die neue Größe des Speichers vom rufenden Programmteil vorgegeben. Entwerfen Sie ein Gerüst für Funktionen, in denen die neue Größe in der Funktion selbst berechnet wird. Neben dem neuen Zeiger müssen diese Funktionen dem rufenden Programmteil auch die neue Feldgröße bekannt machen.

1.4.4 Die Größe von Feldern

Bei der Deklaration von Feldern (*Vektoren*) oder der Anforderung von Speicherbereichen wird dem Speicherbereich eine bestimmte Größe zugeweisen, die nicht

überschritten werden darf. Die Folge von Verletzungen der Indexgrenzen ist Datenunfug in den das Feld umgebenden Variablen, bei Zeigern meist Ursache von Speicherlecks (*siehe* Kap. 3.1).

Eine häufige Ursache von Indexüberschreitungen innerhalb von Funktionen ist die Verwendung von Zahlen zur Deklaration der Feldgröße anstelle von Symbolen:

```
int feld[100];
...
for(i=0;i<100;i++)
   feld[i]=0;
...
```

Stellt sich bei einer Wiederverwendung heraus, dass eine andere Feldgröße notwendig ist, werden die assoziierten Zahlen im Code schnell übersehen (*oder sogar zu viele geändert*). `int feld[50]` führt zu unvollständiger Bearbeitung, `int feld[500]` zu Indexfehlern, wenn die Änderung in der `for` -Anweisung übersehen wird. Sinnvoll ist daher das Arbeiten mit symbolischen Größen:

```
static const int flen = 100;
...
int feld[flen];
...
for(i=0;i<flen;i++)
   feld[i]=0;
...
```

ermöglicht Änderungen an einer zentralen Stelle und verhindert so die beschriebenen Fehler.[33]

Diese Maßnahme bezieht sich auf den Code einer Funktion oder eines Moduls. Werden Felder an Funktionen übergeben, so können diese spätestens nach einer Umstrukturierung des Kodes (*Verschieben in ein anderes Bibliotheksmodul*) nichts mehr über irgendwelche Größenfestlegungen wissen. Zur Übergabe eines Feldes

[33] #definemakros sind eine alternative Möglichkeit. Bei der Entscheidung für die eine oder andere Variante ist der Geltungsbereich der Größenfestlegung ausschlaggebend. #definemakros besitzen eine globale Gültigkeit zwischen ihrer Deklaration und dem vom Compiler verarbeiteten Modulende (*was bei einer Definition in einer Header-Datei zu einigem Durcheinander führen kann, wenn diese in verschiedene andere Header-Dateien eingebuden wird*), Variablen-Deklarationen lassen sich auf ein Modul, eine Funktion oder einen Teil davon beschränken. C–Programmierer setzen aus Gewohnheit meist Makros ein; im Rahmen der besseren Kontrolle möchte ich Ihnen jedoch in C++-Code in Gemeinschaft mit anderen Autoren empfehlen, so weit als möglich auf die Verwendung von Makros zugunsten von typgebundenen Vereinbarungen oder templates zu verzichten.

gehört daher immer die Übergabe der Feldlänge, und wir vereinbaren sinngemäß für den Import oder Export von Variablen:

Regel.

Importtyp	Größenregelung
`int func(char * s)`	Die importierte Größe des Datenbereichs ist genau strlen(s), das heißt der Datenbereich endet bei s[k]==0
`int func(int * l)`	Es wird genau ein Integerwert indiziert
`int func(int * l,int len)`	Es wird ein Feld von Integerwerten bezeichnet, das genau die Länge len besitzt

Exporttyp	Größenregelung
`char * func()`	Die exportierte Größe des Strings ist genau strlen(..) Bytes
`int * func(int *len)`	Die exportierte Länge des Integerfeldes ist in len angegeben. Die Variable len muss vom rufenden Programm zur Aufnahme der Länge bereitgestellt werden.
`char func();`	Einzelne Werte sind stets als Wert-,
`int func();`	nicht als Zeigerrückgaben zu organisieren.
`struct st * func();`	Es wird ein Zeiger auf eine zusammengesetzte einzelne Zeigervariable (Struktur, Klassenobjekt) erzeugt.

„Genau“ bedeutet, dass bereits die Speicherstelle `s[strlen(s)+1]` nicht mehr für `s` reserviert wurde und zu einer anderen Variablen gehört, deren Inhalt nun zerstört wird.[34] Beachten Sie hierbei die Sonderstellung von Stringvariablen: Diese enthalten implizit eine Längenangabe, die nicht mit der tatsächlich reservierten Länge übereinstimmen muss, die wir jedoch als verbindlich definieren. Eine zusätzliche Längenangabe ist nicht notwendig;

- wird umgekehrt aber eine Variable des Typs `(char*)` mit einer Längenangabe übergeben, so ist dies als ein Feld von vorzeichenbehafteten Bytes zu interpretieren, aber nicht als String!

[34] Das gilt auch, wenn der Programmierer es im Augenblick des Programmschreibens besser weiß und der Speicher doch zu *s* gehört. Wie es um solches „Wissen“ bestellt ist, lässt sich leicht feststellen, wenn der Code nach drei Monaten Pause erneut betrachtet wird.

Wir vertiefen dies noch an einigen Beispielen. Unkorrekt (*aber oft genug zu beobachten*) ist

```
int main(int argv,                      Das Betriebssystem sagt
        char ** argc){                  ausdrücklich: „ein
    ...                                 Übergabestringzeiger !". Wo
    if (argv==1){                       soll also der Platz für
      argc[1]=                          einen zweiten herkommen?
          (char*)malloc(80)
    ...
int func(char * s){                     Der übergebene String
    char * rep = keine \                besitzt maximal die Länge 6.
        Antwort;                        Wie soll er einen String der
    ...                                 Länge 14 aufnehmen können?
    if (strlen(s) < 5){
      strcpy(s,rep);
}//endif
...
int main(int argv, char ** argc){       Formal zunächst korrekt,
  argc[0]=(char*)malloc(...);           aber woher weiß man nun am
                                        Programmende, wem der
                                        Speicher gehört, auf den
                                        argc[0] verweist?
```

Aufgabe. Schreiben Sie ein Programm zum Kopieren von Dateien. Die Dateinamen werden entweder beide in der Kommandozeile übergeben oder sind im Programm durch Dialog zu ermitteln. Eine Kopie wird nur erstellt, wenn die Zieldatei noch nicht existiert.

Für diese Erledigung dieser Aufgabe müssen Sie korrekte Versionen des gerade dargestellten Unfugs implementieren. Sinnvoll ist weiterhin die Verwendung von goto und einigen definierten Makros.

Bei Verwendung importierter Strings zu Schreibzwecken ist zwingend eine Längenprüfung vorzunehmen und zweckmäßigerweise eine lokale Variable zur Aufnahme der übergebenen Länge anzulegen. Bei Schreiboperationen innerhalb der Funktion lässt sich so leicht kontrollieren, wie viel Speicherplatz nach Stringverkürzungen erneut belegt werden darf, wobei die abschließende ursprüngliche Null in keinem Fall überschrieben werden darf. Die folgenden Beispiele sind somit nach den Konventionen korrekt, auch wenn man am logischen Sinn ohne weitere Kenntnis der ausgelassenen Kodeteile auf den ersten Blick Zweifel haben darf.

Die Hauptfunktion darf, da sie die angeforderte Größe kennt, im vollen Umfang auf das Feld zugreifen, also auch auf Positionen, die nach Aufruf der Unterfunktion

Hauptfunktion	gerufene Funktion

`char * s =0 ;`
`int len_s;`
`...`
`s = (char*) malloc(len_s);`
`...`
`func(s);`
`...`
`if (i<len_s)`
`  s[i]='A';`
`...`
	`int func(char * s){`
	`  int s_len;`
	`  char txt[3] = "//";`
	`  s_len=strlen(s);`
	`  ...`
	`  if (s_len>2)`
	`  strcpy(s,txt);`
	`  ...`
	`  if (i<s_len)`
	`  s[i]='p':`
	`  ...`

hinter dem offiziellen Stringende liegen, wie in diesem Beispiel möglich. Die gerufene Funktion unterliegt den Einschränkungen der Regeln.[35] Im umgekehrten Fall, dem Export eines in einer Funktion erzeugten Strings, gilt dies sinngemäß auch: Die übernehmende Funktion darf nur bis zum festgestellten Stringende Schreiboperationen durchführen, selbst wenn die erzeugende Funktion ein vielfaches dessen reserviert hat (*auch hier empfiehlt sich bei komplexeren Abläufen die Definition einer Längenvariablen*). Als wichtige Konsequenz ist noch zu vermerken, dass Strings nicht ohne Initialisierung an Funktionen übergeben werden dürfen, da sonst das Stringende nicht festliegt. Sicherheitshalber ist eine Initialisierung als Leerstring bei der Deklaration sinnvoll

```
char * feld[100] =;
char * f2=0;
...
f2=(char*) malloc(100);   // ggf. diese Sequenz als
f2[0]=0;                  // Makro definieren
```

Zweifelsfälle, ob eine Initialisierung nach bedingten Verzweigungen tatsächlich stattgefunden hat, werden hierdurch vermieden.

1.4.5 Pufferüberläufe

Es gilt nicht nur, die Indexgrößen von Feldern nicht eigenmächtig zu ver-letzen, genauso wichtig ist eine Kontrolle, wenn die Größe eines Feldes bekannt ist. Betrachten wir das an einem Beispiel. In der Datenstruktur

[35] In anderen Programmiersprachen wird die Stringlänge anders festgelegt, zum Beispiel durch einen separaten Eintrag, der als Länge zu interpretieren ist (Pascal). Es gelten jedoch die gleichen Einschränkungen: Über die übergebene Länge hinaus ist ein Feld nicht definiert und darf nicht genutzt werden.

```
struct DataSet {
    int   no;
    char  s[40];
    int   ref;
} dataSet;
```

ist ein Stringfeld von 40 Byte Größe angelegt. Die Methode

```
void SetString(const char* s){
    strcpy(dataSet.s,s)
}//end function
```

überträgt Daten in dieses Stringfeld. Sofern sich der Nutzer der Funktion an die Regeln hält, also beachtet, dass der String nicht mehr als 39 gültige Zeichen umfasst, treten keine Probleme auf. Gibt er jedoch längere Strings an, überschreibt die Methode auch die Daten hinter dem Stringfeld.

Regel. Ein solches Verhalten ist natürlich nicht akzeptabel, und daher notieren wir als Programmierregel: In extern nutzbaren Funktionen mit einem Zugriff auf intern begrenzte Felder muss eine Längenkontrolle bei Schreibzugriffen stattfinden.

Die korrekte Implementation der obigen Methode nach dieser Regel ist

```
void SetString(const char* s){
    dataSet.s[39]=0;
    strncpy(dataSet.s,s,39)
}//end function
```

Dieser Fehler – Kontrollen unterlassen und stillschweigend davon ausgehen, dass sich der Nutzer ebenfalls an Konventionen hält – tritt häufiger und mit größerem Schaden auf, als man vielleicht im ersten Augenblick vermutet. Nahezu alle Probleme in Internetprotokollen, die mit aktivem Eindringen in fremde Rechnersysteme über das Netzwerk verbunden sind, sind auf derartige fehlende Kontrollen zurückzuführen.

1.4.6 Importverwendung

Die Parameterübergabe als Wert oder als Referenz in Funktionsköpfen hat zwei Funktionen:

- In der Bibliotheksfunktion[36] soll bei Wertübergabe die Möglichkeit ausgeschlossen werden, bei der Programmierung des Funktionskörpers „versehentlich“ wesentliche Daten des Auftraggebers zu verändern.

[36] Eine Programmierung einer Funktion sollte grundsätzlich unter der Annahme erfolgen, daß sie später in eine Bibliothek eingegliedert wird. Damit wird eine Laxheit bei der Erstellung von „Einmal-Hilfsroutinen“ vermieden.

- Dem Auftraggeber, das heißt dem Benutzer der Bibliotheksfunktion, soll durch die Schnittstellenbeschreibung verbindlich signalisiert werden, ob er mit einer Konstanz oder einer Veränderung seiner Daten zu rechnen hat, um entsprechende Schritte vor Benutzung der Funktion durchzuführen.

1.4.6.1 Konstante Referenzen

Bei der Übergabe von Feldern als Funktionsparameter ist häufig aus Geschwindigkeits- und Speicherplatzgründen eine Übergabe als Referenz (*Zeiger*) notwendig.[37] Dabei sollte das funktionale Konzept nicht aufgegeben werden:

Funktionskopf	Aufrufende Instanz	Bibliotheksprogrammierer
`int func(int li)`	Der Wert der Variable li wird nicht verändert	Der Wert von li darf beliebig verändert werden
`int func(int *li)`	Der Wert der Variablen li wird voraussichtlich verändert	Der Wert von li darf beliebig verändert werden
`int func( const int *li, int len)`[38]	Das Feld li wird trotz Übergabe als Zeiger nicht verändert	Der Übersetzer verhindert Veränderungsversuche des Bibliotheksprogrammmierers. Sind im Programmablauf Änderungen notwendig, so müssen die betreffenden Teile von li auf eigene lokale Variable umkopiert werden.

Anmerkung. Eine Fehlerquelle in C-Programmen ist, dass die Schnittstellendefinition eines Typs im Funktionskörper und im Aufruf beachtet werden muss, ein Zeiger also immer als Zeiger angesprochen und eine normale Wertvariable beim Aufruf in einen Zeiger überführt werden muss. Sofern die Verwendung von C++ möglich ist, ist die Verwendung des Referenzkonzeptes zu empfehlen, das von der Beobachtung des genauen Typs entlastet. An die Stelle der angegebenen Funktionsaufrufe tritt dann

[37] Im allgemeinen können Strukturen auch als Wertparameter übergeben werden, jedoch trifft das nicht unbedingt auf den Inhalt aller Felder der Struktur zu (*siehe unten*). In C können Felder nur als Zeiger übergeben werden. Ein „Umweg“ für statische Felder ist die Einbindung in eine Struktur.

[38] Zur Beachtung: const int $*$ macht nur bei Feldern Sinn! Deshalb eine Längenangabe.

```
int func1(T& i);    // Wert kann verändert werden
int func2(T const& t);    // Wert konstant
// Der Aufruf erfolgt immer als Wertübergabe:
int i, *j;
...
func1(i); func2(*j);
```

In zwei Fällen ist bei der Implementation von Bibliotheksfunktionen besondere Vorsicht angebracht. Bearbeitet die Funktion Felder, so ist folgende Schnittstellendefinition nicht selten:

```
int func(double* d_out,
         double const* d_in, int dlen);
```

Die Trennung von Ein- und Ausgebdaten kann den Anwender jedoch nicht an folgender Verwendung hindern:

```
...
res=func(data,data,len);
...
```

Das Ergebnis wird auf dem Feld mit den Eingabedaten wieder erwartet. In der Funktion ist das durch Vergleich der Zeigeradressen feststellbar, und als Konsequenz muss in einem solchen Fall ein Ergebniswert so lange auf einer Hilfsvariablen gespeichert werden, bis die entsprechende Position auf dem Feld `d_in` für die Verarbeitung nicht mehr benötigt wird.[39] Dabei ist auf sparsamen Umgang mit den Systemressourcen zu achten: Eine Zeile einer 5.000×5.000-Matrix zu speichern ist etwas anderes als die komplette Matrix zu duplizieren, und diese Strategie ist beispielsweise hinreichend, um eine Matrizenmultiplikation der Art `mulmat(a,a,b,len)` durchzuführen.[40] An dieser Stelle sind häufig wieder solide mathematische Kenntnisse gefordert, um die Algorithmen entsprechend zu analysieren.

1.4.6.2 Besonderheiten bei Strukturen

Der andere Problemfall betrifft Strukturen mit Feldern als Attributen. Auch sebstdefinierte Datentypen (*Strukturen*) können in Funktionsparametern als Werte oder als Zeiger übergeben werden. Wir betrachten dazu folgende Fälle:

---

[39] Falls Sie einmal eine Programmiersprache verwenden, die keine Adressprüfung vorsieht, können Sie auch folgendermaßen vorgehen: Verändern Sie temporär den Wert eines Elements auf der Ausgabevariablen. Ist die gleiche Veränderung auch auf der Eingabevariablen zu beobachten, handelt es sich wohl um die gleiche Variable. Diese Vorgehensweise ist sicherer als der Vergleich der Inhalte der Felder (wenn die nicht gleich sind, ist natürlich auch alles erledigt, aber hier genügt die Prüfung eines Elements).

[40] Das funktioniert auch noch bei mulmat(a,b,b,len), aber nicht mehr bei mulmat(a,a,a,len). Warum nicht? Vergleiche auch Kapitel 5.

Feste Feldgröße	Variable Feldgröße

`struct A {`
`  int m[10];`
`  double r[10];`
`  int len;`
`} a;`
`a.len=10;`
`...`
	`struct B {`
	`  int * m;`
	`  double * r;`
	`  int len;`
	`};`
	`...`
	`struct B b ;`
	`b.m = (int*)`
	`malloc(10*sizeof(int));`
	`b.r = (double*)`
	`  malloc(10*sizeof(double));`
	
	`b.len=10;`
	`...`

Abgesehen vom Aufwand bei der Erzeugung und Vernichtung und dem zusätzlichen Längenfeld in der Struktur B können beide Strukturen auf die gleiche Art genutzt werden. Bei der Übergabe als Wertparameter in eine Funktion existieren jedoch markante Unterschiede:

```
int func (A a)                    int func( B a)
...                               ...
a.r[1] = 3; // Im                 a.r[1] = 3; // Auswirkung
Hauptprogramm                     im
            // nicht                        // Hauptprogramm
sichtbar
```

- Struktur `A` wird komplett kopiert, das heißt alle in den Teilfeldern `m` und `r` vorhandenen Daten stehen als Kopie in der Funktion zur Verfügung und dürfen verändert werden.
- Bei Struktur `B` werden nur die Zeigeradressen kopiert, die Datenfelder `m` und `r` sind die gleichen wie im rufenden Programmteil, das heißt Änderungen finden sich anschließend auch im rufenden Programm wieder.

Hilfen für die Vermeidung von Fehlern bei der Programmierung der Funktionen stellt C leider nicht zur Verfügung. Auch eine `const`–Vereinbarung wird vom Übersetzer „übersehen“:
Das Ausbleiben einer Fehlermeldung ist hier korrekt: `const` bezieht sich auf die Felder der Struktur, und diese enthalten Zeigeradressen. Ein Versuch, diese zu ändern, wird vom Übersetzer geahndet. Die Meldung im linken Beispiel nützt hier nichts, da es sich ja um eine Kontrolle bezüglich versehentlicher Fehler handelt, und die Mühe, in solchen Fällen zunächst fest dimensionierte Strukturen zu vereinbaren,

Prüfung des Inhalts	Prüfung der Adresse

`int func (const A* a)`
`...`
`a->r[1] = 3;`
`  //Übersetzerfehler`
	`int func(const B* a)`
	`...`
	`a->r[1]=3; //keine`
	`  // Fehlermeldung!`

aufgrund der damit verbundenen Fehlermöglichkeiten ziemlich unsinnig ist. Da man daher in einer Funktion in der Regel nicht sicher weiß, was drumherum so passiert, sollte man sicherheitshalber von zeigerhaltigen Strukturen ausgehen (*was häufig anwendungstechnisch auch sinnvoller ist*) und selbst verstärkt auf Zuweisungsfreiheit achten anstatt auf Unterstützung des Übersetzers zu bauen.

Der Vollständigkeit halber sei auf Fehlerquellen hingewiesen, die auf falschem Gebrauch von Sprachkonstrukten beruhen:

```
int func(const char* h){
    char * c;
    c=const_cast<char*>
        (h);
    ...
```

```
int func(
    const MyClass& obj){
      MyClass& h=
        const_cast<MyClass&>
            (obj);
    ...
```

Beide Deklarationen und Anweisungen hebeln die const-Vereinbarung in der Funktionsschnittstelle auf und erlauben Änderungen auf dem als konstant definierten Objekt. Auch wenn das technisch sinnvoll sein mag (*das Objekt wird temporär verändert, liegt aber bei Verlassen der Funktion wieder im Originalzustand vor*) – sofern die übergebenen Objekte tatsächlich Konstante sind, bricht das Laufzeitsystem die Anwendung mit einer Schutzverletzung bei einem schreibenden Zugriff ab. Solche Anweisungen dürfen daher nur dann verwendet werden, wenn der Charakter der umgewandelten Variablen genau bekannt ist.

Die beschriebenen Effekte sind dann zu be(ob)achten, wenn der Programmaufbau der C–Notation folgt. Vieles entschärft sich, wenn die strengeren Kontrollmechanismen der C++ - Notation eingesetzt werden, also beispielsweise die `cast`-Operatoren, die zumindest versehentliche Umwandlungen verhindern, sowie Klassen mit gut definierten Zugriffsmethoden und geschützten Attributen anstelle von einfachen Strukturen. Wir wir im folgenden Kapitel darlegen werden, muss die Verwendung von Klassenkonzepten nicht unbedingt zu Einbußen in der Geschwindigkeit führen. Allerdings ist der Programmieraufwand natürlich höher: Das Klassendesign will gut überlegt sein, die Programmierung der Schnittstellen und ihre Implementation ist zwar einfacher, erfordert aber aufgrund der reinen Textmenge ihre Zeit. Der Rückgriff auf „einfachere“ Mechanismen ist deshalb oft verständlich, und wir wissen ja jetzt, worauf dabei zu achten ist.

1.4.6.3 Speichergrößen von Strukturen

Schließen wir die Einführung durch zwei letzte Bemerkungen zu Strukturen ab. Für die Erzeugung und Vernichtung von Speicherplatz für Strukturen gilt sinngemäß das bereits bekannte:

```
struct A * a = 0;
...
a = (A*) malloc(n*sizeof(A));
...
free(a);
```

Aus Gründen der Rechnerarchitektur gilt für die rechnerische und die tatsächlich reservierte Speichergröße die Relation

$$sizeof(A) \geq \sum_{attributes} sizeof(A.attribute)$$

Bei der Bereitstellung von Speicherplatz für Zeigervariablen auf Strukturen oder bei der Kopie des Wertebereiches einer Strukturvariablen auf den Wertebereich einer anderen ist daher sizeof `(struct_identifier)` zu verwenden (*es sei denn, man kopiert die Attribute einzeln*). Umgekehrt ist es bei der Sicherung von Strukturen in Dateien für den Austausch mit anderen Rechnern nicht garantiert, dass die Funktion

```
fwrite(a,sizeof(A),1,file);
...Transport auf einen anderen Rechner ...
fread(a,sizeof(A),1,file);
```

korrekt funktioniert. Hier muss mit den einzelnen Attributen operiert werden, zum Beispiel

```
fwrite(&a.i,sizeof(int),1,file);
fwrite(&a.ch,sizeof(char),1,file);
fwrite(&a.r,sizeof(double),1,file);
...Transport auf einen anderen Rechner ...
fread(&a.i,sizeof(int),1,file);
fread(&a.ch,sizeof(char),1,file);
fread(&a.r,sizeof(double),1,file);
```

Auch das muss aber nicht in jedem Fall zum Erfolg führen, da immer noch unterschiedliche Typen oder Darstellungen implementiert sein können. Im Zweifelsfall sind die Schnittstellenbeschreibungen der Datentypen auf den Maschinen zu konsultieren.

1.4.7 Operatorenverwendung

Mit dem Übergang von C zu C++, der anstelle eines Übergangs von einer prozeduralen zu einer objektorientierten Betrachtungsweise zunächst einmal auch als verstärkte Einbindung des Übersetzers in die Kontrolle der korrekten Ausweitung von Algorithmen auf komplexere algebraische Strukturen aufgefasst werden kann, kommt, wie in der Einführung schon erwähnt, die Möglichkeit des Überschreibens von unären `{-, !, ++, --, ~ }` oder binären Operatoren `{+, -, *, /, %, <, >, ==, >=, <=, !=, &, |, ^, &&, ||, <<, >> }` sowie von Indexoperatoren `{[ ], ( ) }`und einigen weiteren hinzu.

Das hat Vorteile und Nebenwirkungen (*ich vermeide bewusst den Term „Nachteil“*). Oftmals liegen Algorithmen für bestimmte Datentypen bereits vor, und für eine Verwendung mit neuen Typen muss nur die Typbezeichnung im Deklarationsteil ausgetauscht werden. Längere Umstellungsarbeiten, wie sie beim Übergang von reellen zu komplexen Zahlen in C auftreten, da alle Rechenoperationen durch Funktionsaufrufe ersetzt werden müssen, sind nicht notwendig, die damit verbundenen Fehlerquellen entfallen, und auch das Herumärgern mit leicht geänderten Funktionsnamen oder Aufrufkonventionen, wenn ein weiterer Datentyp zur Anwendung kommen soll entfällt. Bei der Implementation von neuen Algorithmen kann im mathematischen „Jargon“ ohne Umweg über die Auswahl der zugehörenden Funktionen programmiert werden. Die Nebenwirkung besteht meist darin, bei der Implementation eines Datentyps sehr viel Aufwand treiben zu müssen, um alle Fälle abzudecken.[41]

Um dem Compiler die Kontrollaufgaben zu ermöglichen, ist bei einer Implementation die mathematische Bedeutung der Operatoren genau abzubilden. Läßt man hier die notwendige Konsequenz vermissen, so wird dies entweder irgendwann durch Rechenfehler belohnt, oder man lernt die Konsequenz des Compilers kennen, der bestimmte Kodes einfach nicht mehr übersetzen will, obwohl „doch alles richtig ist“. Gehen wir der Sache auf den Grund. Die Rechenvorschrift

$$a = b + c$$

erwartet ein Ergebnis in der Variablen a, während die beiden Summanden `b,c` unverändert aus der Operation hervorgehen. Bei Überschreiben der Operatoren laufen formal folgende geschachtelte Funktionsaufrufe ab:

```
a.operator=( b.operator+(c) );
```

Der Variablen `b` „gehört“ somit der Operator „+“, das Ergebnis des Funktionsaufrufs ist Übergabeparameter des überschriebenen und `a` „gehörenden“ Operators „=“.

[41] Dabei handelt es sich tatsächlich meist um eine Nebenwirkung. Der Aufwand bei C–Vorgehensweise ist im Grunde nicht geringer, wird aber gefühlsmäßig der Anwendung und nicht der Typimplementation zugeschrieben und wirkt dadurch geringer.

Da `b,c` konstant sind, ist der Rückgabewert von `operator+` zwangsweise eine temporäre Variable. Die korrekte Schnittstellendefinition der Addition lautet somit

```
T operator+(T const& op) const;
```

Anstelle einer konstanten Referenz auf `op` im Aufruf könnte natürlich auch eine Wertübergabe erfolgen; auf die Vorteile einer Zeigerübergabe bei größeren Datenstrukturen muss aber wohl nicht weiter eingegangen werden. Wichtig sind beide `const` -Vereinbarungen, da ja beide Summanden konstant bleiben sollen.

Bei der Zuweisung erinnern wir uns an die Möglichkeit verketteter Zuweisungen in C, die als verketteter Funktionsaufruf in folgender Form beschrieben werden können:

```
a=b=c   <=>   a.operator=( b.operator=(c) )
```

Der Operand rechts der Zuweisung bleibt konstant. Mit dem bereits bei der Addition festgestellten Vorteil einer Zeigerübergabe erhalten wir die Schnittstellenvereinbarung

```
T&  operator=(T const& op)
```

In entsprechender Weise können die anderen Rechenoperatoren {`-,*,/,%`} (*letzterer natürlich nur für Ringe und nicht mehr für Körper*) implementiert werden. Die Kombinationsoperatoren {`+=,-*,*=,/=,%=`} fallen in die gleiche Kategorie wie der Zuweisungsoperator. Sofern für die betrachtete Klasse ebenfalls sinnvoll, fallen auch die Schiebeoperatoren in diese Kategorien.

Unäre Operatoren lassen definitionsgemäß das bezogene Objekt konstant und liefern ein neues temporäres Objekt, das heißt die korrekte Schnittstelle ist

```
T operator-() const;
```

Inkrement- und Dekrementoperatoren sind danach zu unterscheiden, ob das Inkrement in der folgenden Operation verwendet wird oder der Ausgangswert und das Inkrement nur gespeichert wird. Im ersten Fall wird inkrementiert und eine Referenz erzeugt, im zweiten Fall ein temporäres Objekt erzeugt:

```
++a:    T&  operator++()
a++:    T   operator++(int)
```

Es ist also nicht egal, welche Form des Operators verwendet wird, wenn *T* kein simpler Standardtyp ist. Entgegen der eingebürgerten Verwendung von `a++` in Schleifenkonstrukten sollte aus Effizienzgründen `++a` verwendet werden.

Bei der Definition von Vergleichsoperatoren kann man sich auf

```
bool operator==(T const& op) const
bool operator< (T const& op) const
```

beschränken. Die weiteren logischen Operatoren lassen sich nämlich durch die `template` -Funktionen (*siehe auch nächsten Kapitel*)

```
template <class T1, class T2>
inline bool operator!=(const T1& x, const T2& y){
    return !(x==y);
}//end function

template <class T1, class T2>
inline bool operator>(const T1& x, const T2& y){
    return y<x;
}//end function

template <class T1, class T2>
inline bool operator<=(const T1& x, const T2& y){
    return !(y<x);
}//end function

template <class T1, class T2>
inline bool operator>=(const T1& x, const T2& y){
    return !(x<y);
}//end function
```

allgemein gültig implementieren.

Indexoperatoren werden als Referenzen implementiert, das heißt auf die Variablen wird direkt zugegriffen. Tritt der Zugriff an einer Stelle auf, an der eine Kopie des Wertes erzeugt werden muss, so für der Übersetzer dies anschließend aus.

```
a[i]=b[k];        ===> entspricht
   T& operator[](int) =
            T const& operator[](int i) const;
a(i,j)=b(k,l);    ===> entspricht
   T& operator()(int,int) =
            T const& operator()(int,int) const;
```

Der Index-Operator mit eckigen Klammern akzeptiert gemäß Sprachstandard nur einen Parameter. Sind mehrere Indexparameter notwendig, ist ein Index-Operator mit runden Klammern zu verwenden, wobei hier die Anzahl der Parameter nicht beschränkt ist.

Beim Referenzoperator ist in einigen Fällen Vorsicht geboten. Bei einem Indexzugriff, der eine Referenz zurück liefert, lässt sich beispielsweise nicht mehr kontrollieren, welcher Wert abgespeichert wird. Die Funktion ist bereits verlassen, bevor der Wert abgelegt wird. Betrachten wir vorab als Beispiel ein Polynom.[42] Der größte Index der von Null verschiedenen Koeffizienten definiert den Grad eines Polynoms. Konkret könnte beispielsweise der Wert auf Null gesetzt und dadurch

[42] Wir werden eine Polynomklasse in Kapitel 9.5 noch genauer untersuchen und beschränken uns hier mehr oder weniger nur auf einige Stichworte.

der Grad des Polynoms erniedrigt worden sein. In einer Polynomklasse wird man sich aber aus Effizienzgründen darauf verlassen wollen, dass jede Funktion das Objekt mit definiertem Grad verlässt. Anstelle des oben angegebenen Operators ist das Methodenpaar

```
T const& operator[](int i) const;
void set(int index, double value);
```

zu definieren. Die Zuweisung kann dann nicht mehr übersetzt werden und ist durch folgende Zeile zu ersetzen:

```
a.set(i,b[k]);
```

Effekte dieser Art treten bei Klassen auf, deren Attribute voneinander abhängig sind und die Änderung eines Attributs die Änderung eines anderen nach sich ziehen kann. Sie sind leicht beherrschbar, wenn die Entwickler über den notwendigen theoretischen Hintergrund für den Umgang mit der Materie verfügen, die sie mit ihrem Code behandeln. Ich möchte hier ausdrücklich feststellen, dass eine korrekte Behandlung an diesen Kenntnissen hängt. Ein grundsätzlich Festlegung, immer die `set()`–Methode einzusetzen, nützt nichts, da ohne Kenntnisse auch das nicht korrekt implementiert würde, und ich lehne sie daher ab.

Ein nicht sehr häufiger, aber durchaus auftretender Anwendungsfall ist die Notwendigkeit, ein Attribut in einer als `const` definierten Methode vorübergehend zu ändern (*bei Verlassen der Methode muss/sollte natürlich wieder der Anfangszustand hergestellt sein*). Ein entsprechender Versuch ist natürlich zum Scheitern verurteilt, da der Übersetzer dem Programmierer gnadenlos auf die Finger schlägt. Das Streichen der `const`–Definition ist allerdings das Dümmste, was einem hier einfallen kann. Man kann sich schnell ein Bild von den Folgen einer solchen Aktion machen, wenn man in einer relativ komplexen Anwendung einmal spaßeshalber ein `const` streicht: In allen weiterhin als `const` definierten Methoden, in denen die geänderte Methode verwendet wird, hebt ein großes Compilergeschrei an. Das Problem muss in der Klassendefinition behoben werden, wozu C++ ein Schlüsselwort zur Verfügung stellt

```
mutable T att;
```

mutable sorgt für ein Aussetzen der `const`–Überprüfung für dieses Attribut. Problem gelöst, wie bei allen anderen Tricks sollte man sich aber immer darüber im Klaren sein, dass man die neuen Möglichkeiten auch bezahlen muss, hier durch eine geringere Kontrolle durch den Übersetzer