Objektorientierte Programmierung
(Winter 2010/2011)

Kapitel 10: Bessere Programmierung (Guter Stil)

- Vermeidung und frühzeitige Erkennung von Fehlern
- Konstanten, Aufzählungstypen, `typedef`
- Zusicherungen (`assert`)
- Warnungen des Compilers

Inhalt

Allgemeines

Anforderungen an gute Programme:

- Korrektheit (Fehlerfreiheit)
- Verständlichkeit
- Sicherheit
- Effizienz
- Systemunabhängigkeit / Portabilität
- Leichte Anpassbarkeit an veränderte Bedingungen
- Benutzerfreundlichkeit

Korrektheit (1)

- Natürlich ist die grundlegendste Forderung an ein Programm, dass es die gegebene Aufgabe erfüllt, d.h. die gewünschte Funktion von Eingaben auf Ausgaben berechnet.

  Eingaben und Ausgaben sind hier etwas weiter zu sehen, und schließen auch Inhalte von gelesenen oder geschriebenen Dateien ein, sowie Daten, die über das Netzwerk empfangen oder gesendet werden.

- Insbesondere sollte ein Programm nicht abstürzen, d.h. einen Hardwarealarm auslösen (z.B. Division durch 0, "Segmentation Fault"), oder in eine Endlosschleife geraten.

Korrektheit (2)

- Wenn man nur einen Teil eines Programms erstellen soll (etwa eine Funktion oder eine Klasse), ist die Einhaltung der Schnittstellen eine wesentliche Korrektheitsforderung.

  D.h. die Funktionen sollten die Parameter- und Ergebnistypen haben, die sich aus der Aufgabenstellung ergeben, es sollte keine zusätzlichen Ausgaben oder globalen Variablen geben. Anfänger verwechseln oft Eingaben der Funktion über Parameter mit Eingaben von der Tastatur (`cin`), und entsprechend Rückgabewerte von Funktionen mit Ausgaben auf den Bildschirm. Oft ist aber ein Beispiel-Aufruf oder Testprogramm gegeben, mit dem man diese Frage leicht entscheiden könnte. Nicht selten benutzen Anfänger auch globale Variablen anstelle von Parametern, was ebenso der Aufgabenstellung widerspricht, und beim Übersetzen bzw. Linken des Programms zu Fehlern führt.

Korrektheit (3)

- Man sollte vermeiden, Beschränkungen in das Programm einzubauen, wie z.B., dass Eingaben nur bis zu einer gewissen Größe verarbeitet werden können.

  Es sei denn, das in der Aufgabenstellung explizit erlaubt. Natürlich muss man immer den Aufwand mit dem Nutzen abwägen — eventuell kann man die Aufgabenstellung ändern (oder in der Klausur fragen). Strenggenommen stellt aber jede in der Aufgabenstellung nicht genannte Beschränkung eine Verletzung der Korrektheit dar. Lässt sich die Beschränkung nicht mit vertretbarem Aufwand vermeiden, muss mindestens eine anständige Fehlermeldung ausgegeben werden. Außerdem muß die Beschränkung dokumentiert werden. Oft kann man wenigstens erreichen, dass die Grenze, bei der die Beschränkung greift, leicht zu ändern ist (etwa über eine Konstante im Programm). Das Programm muss dann neu übersetzt werden.

Korrektheit (4)

- Korrektheit wird wahrscheinlicher durch:
  - ◇ Gute Planung, Nachdenken vor und während der Programmierung

    Es kommt auf die Einstellung zur Programmierung an: Ziel sollten 100% korrekte Programme sein. Man sollte nicht annehmen, das alle wichtigen Fehler beim Testen noch gefunden werden, und ein paar übriggebliebene Fehler normal sind. Übrigens verursachen Fehler, die erst beim Testen gefunden werden, am Ende viel mehr Aufwand, als wenn man bei Planung oder Programmierung etwas länger nachgedacht hätte.
  - ◇ Verständlichen, übersichtlichen Programmcode
  - ◇ Gute Dokumentierung

    Z.B. Voraussetzungen einzelner Programmteile.

Korrektheit (5)

- Korrektheit wird wahrscheinlicher durch (Forts.):
  - ◇ Ausnutzung aller Prüf-Möglichkeiten des Compilers: Typisierung, `const`-Deklarationen, möglichst viel `private`/lokal, Warnungen anschalten.

    Ein Ziel ist hier, nur das möglich zu machen, was benötigt wird und beabsichtigt ist. Wenn man die Chance hat, dass der Compiler bestimmte Arten von Fehlern bereits findet, sollte man sie nutzen: Sie können nicht beim Testen durchschlüpfen und die Beseitigung macht weniger Arbeit, als wenn man sie erst mit dem Debugger suchen muss.
  - ◇ Prüfung im Programmcode selbst (z.B. `assert`).

    Siehe Abschnitt ab Folie 10-59.

Korrektheit (6)

- Korrektheit wird wahrscheinlicher durch (Forts.):
  - ◇ Defensive Programmierung

    Man verlasse sich in einer Funktion/einer Klasse nicht darauf, dass sie korrekt aufgerufen/benutzt wird, sondern prüfe alle Parameter sowie den Zustand des Objektes.
  - ◇ Kritisches Lesen des Quellcodes durch andere Programmierer (Code Review / Code Audit).
  - ◇ Testen (siehe nächste Folie)
  - ◇ Zusätzliche Ausgaben in Debug-Konfiguration

    Man erleichtert sich das Finden von Fehlern, wenn zusätzliche Testausgaben im Programm bereits vorgesehen sind, und leicht anzuschalten sind (genauer Ablauf, Dump von Datenstrukturen).

Korrektheit (7)

- Zum Testen:
  - ◇ Es müssen mehrere Tests durchgeführt werden, die die unterschiedlichen Fälle abdecken.

    Wenn man z.B. einen Primzahltest programmieren soll, wäre es sehr ungeschickt, wenn man ihn nur mit Primzahlen testet. Man muß positive und negative Fälle testen, also Primzahlen und Nicht-Primzahlen.
  - ◇ Auch die Reaktion auf fehlerhafte Eingaben ist zu prüfen.
  - ◇ Jede Zeile des Programms muß in mindestens einem Test mindestens einmal ausgeführt werden.

Korrektheit (8)

- Zum Testen:
  - ◇ Man achte besonders auf Extremfälle (z.B. wenn eine Schleife überhaupt nicht bzw. genau einmal ausgeführt wird).

    Wie reagiert das Primzahl-Programm z.B. auf 1 oder 2?
  - ◇ Man kann sich Shellskripte schreiben, mit denen man die Tests nach jeder Änderung des Programms wiederholen kann.

    Z.B. kann man mehrere Aufrufe eines Programms in eine Datei schreiben, eventuell auch gleich zusammen mit den Benutzereingaben. Die Ausgabe kann man in eine Datei umlenken und mit `diff` oder `cmp` mit der erwarteten Ausgabe vergleichen.

Korrektheit (9)

- "Debugging is hard and can take long and unpredictable amounts of time, so the goal is to avoid having to do much of it. Techniques that help reduce debugging time include good design, good style, boundary condition tests, assertions and sanity checks in the code, defensive programming, well-designed interfaces, limited global data, and checking tools. An ounce of prevention really is worth a pound of cure." [Kernighan/Pike]

Verständlichkeit (1)

- Der Programmtext (Quellcode) sollte für andere Programmierer möglichst leicht zu verstehen sein, z.B. um
  - ◇ die Korrektheit zu prüfen,
  - ◇ aufgetretene Fehler zu entfernen,
  - ◇ das Programm zu erweitern oder an veränderte Anforderungen anzupassen.
- Es ist eine bekannte Erfahrung, dass man sich in seinem eigenen Programm nicht mehr zurechtfindet, wenn man es nach einiger Zeit ändern muß.

Verständlichkeit (2)

- Verständlichkeit erreicht man u.a. durch
  - ◇ selbstdokumentierende Bezeichner,

    Namen wie `i` und `n` sollte man nur für Laufvariablen in kleinen Schleifen verwenden (die man noch als Ganzes überblicken kann). Ansonsten wären Worte oder Abkürzungen nützlich, die auf die Bedeutung des Wertes in der Variablen schließen lassen.
  - ◇ Kommentare,
  - ◇ zusätzliche Dokumentation,

    Besonders zur globalen Struktur größerer Programme, als Einstieg vor dem Studium des eigentlichen Quellcodes. Aber auch zu besonders komplexen Details, etwa ungewöhnlichen Algorithmen.
  - ◇ einfache Programmstrukturen.

Verständlichkeit (3)

- Zur Kommentierung:
  - ◇ Klassisches Beispiel für überflüssigen Kommentar (der Leser kann selbst C++).

```
i = i + 1; // i um 1 erhoehen
```

  - ◇ Oft wird ein etwas größerer Kommentarblock vor einem Abschnitt des Programms empfohlen, statt jede Zeile zu kommentieren.

    Damit ist der Kommentar auf einer etwas höheren Abstraktionsebene. Außerdem hilft das zum Verstehen der Struktur des Programms.

Verständlichkeit (4)

- Zur Kommentierung, Forts.:
  - ◇ Wenn die Bedeutung von Attributen oder Variablen noch nicht eindeutig aus dem Namen folgt, ist die Deklaration zu kommentieren.
  - ◇ Einschränkungen oder Voraussetzungen müssen dokumentiert werden.

    Z.B. Funktion darf erst nach einer anderen aufgerufen werden.
  - ◇ Im Prinzip muß der Leser einen Korrektheitsbeweis führen können.

    Dazu sind verwendete Bedingungen über die Daten und wichtige Folgerungen aus dem Programmcode zu kommentieren (etwa Bedingung für `else` nach langer `else if`-Kette, Schleifeninvarianten).

Verständlichkeit (5)

- Zu "einfache Strukturen":
  - ◇ Man sollte zu tiefe Schachtelungen von Kontrollstrukturen vermeiden — bei Bedarf kann man ein Stück in eine Prozedur/Funktion auslagern.

    Rückt man abhängige Anweisungen (z.B. Schleifenrumpf) um je einen Tab (8 Zeichen) ein, sollte man mit 79–80 Zeichen/Zeile auskommen, ohne zu viele Statements aufspalten zu müssen.
  - ◇ Die meisten Prozeduren sollten ganz auf den Bildschirm passen ($\leq$ 25 Zeilen).

    Zumindest muß es sonst eine ganz einfache Struktur geben, wie z.B. ein großer `switch`, oder eine einfache Folge von Ausgabeanweisungen. Wenn man bei `if else` den kürzeren Fall zuerst behandelt, kann man die Kontrolle leichter überblicken.

Verständlichkeit (6)

- Zu "einfache Strukturen", Forts.:
  - ◇ In dieser Vorlesung ist `goto` verboten, `break` und `continue` sollten sparsam verwendet werden.

    Anweisungen sollten normalerweise am Anfang betreten und am Ende verlassen werden. Man sollte sich an die bewährten und allgemein bekannten Muster halten, und nicht mittels `goto` eine ganz beliebige Ablaufreihenfolge schaffen ("Spaghetticode").
  - ◇ Im Rumpf einer `for`-Schleife sollte keine Zuweisung an die Laufvariable stehen.

    Man sollte sich darauf verlassen können, dass im Kopf der Schleife die ganze Information über den Ablauf steht, bzw. dass äußerstenfalls ein vorzeitiger Abbruch mit `break` oder `return` erfolgt.

Verständlichkeit (7)

- Zu "einfache Strukturen", Forts.:
  - ◇ Man vermeide unnötige Komplikationen. Oft ist die kürzere Lösung auch die einfachere. Beispiel:

```
if(n > 0)
    return true;
else
    return false;
```

```
int ergebnis;
ergebnis = n * n;
return ergebnis;
```

    bewirken das gleiche wie

```
return n > 0;
```

```
return n * n;
```

  - ◇ Duplizierter Code (z.B. eine getrennte Behandlung der ersten Ausführung einer Iteration) fällt auch in diese Kategorie.

Verständlichkeit (8)

- Man vermeide allzu lange Parameterlisten bei Prozeduren/Funktionen.

  Eine Studie soll ergeben haben, dass Programmierer sich 6 Parameter gerade noch merken können. Mir scheint das schon zuviel. Parameter können reduziert werden durch Übergabe von Objekten, die mehrere Parameter zusammenfassen, oder bei Methoden durch Benutzung von Attributen, die zuerst mit anderen Methoden gesetzt werden. Dann muß man aber einen komplexeren Objektzustand im Kopf behalten, so dass hier genau abgewogen werden muss, was wirklich besser ist.
- Die Reihenfolge mehrerer Parameter sollte nach einem einheitlichen Prinzip gewählt werden.

  Z.B. Eingabeparameter vor Ausgabeparametern. Parameter verschiedener Typen einheitlich anordnen (z.B. immer Typ A vor Typ B).

Verständlichkeit (9)

- Der Leser eines Programms muss für alle Variablen, die noch verwendet werden, im Kopf behalten, welcher Wert gerade dieser Variable steht.

  Man sollte das möglichst erleichtern. Z.B. bei `for`-Schleife gut, dass alle Zuweisungen an die Laufvariable zusammen stehen. Es wäre es günstig, nicht allzu viele Variablen zu benutzen, und die Verwendung nahe an der Zuweisung zu haben. Kommentare, gute Variablennamen!

- Man verwende symbolische Konstanten statt direkten Datenwerten (z.B. `pi` statt `3.14159`).
- Anwendungsspezifische Datentypen statt z.B. `int` können einen Beitrag zur Verständlichkeit leisten.

Sicherheit (1)

- Falsche oder gar bösartige Eingaben sollen mit einer verständlichen Fehlermeldung quittiert werden, und nicht
  - ◇ zum Programmabsturz (Hardwarealarm) oder
  - ◇ zu nicht beabsichtigtem Verhalten führen.
- Hacker nützen Schwachstellen in Programmen aus, so dass die Programme etwas ganz anderes tun, als der Programmierer beabsichtigt hat.

  Das bedeutet natürlich, dass das Programm nicht korrekt ist: Korrektheit verlangt das spezifizierte Verhalten für beliebige Eingaben.

Sicherheit (2)

- Bekannt sind "Buffer Overflows", bei denen auf Arrays außerhalb der Indexgrenzen zugegriffen wird, weil der Programmierer nicht mit extrem langen Eingaben gerechnet hat.
- Wenn das Programm Kommandos aus den Benutzereingaben konstruiert, können unerwartete Zeichen gefährlich sein.

  Z.B. man rechnet mit einem einfachen Dateinamen, und hängt ihn an den Pfad zu einem bestimmten Verzeichnis an, aber der Hacker verwendet `../`, um das Verzeichnis zu verlassen. Oder hängt mit "`; rm -rf ~`" noch ein Kommando an einen mit `system` ausgeführten Befehl an.

Sicherheit (3)

- Zur Sicherheit eines Programms gehört auch, dass selbst in Fällen wie einem Stromausfall oder einem Absturz des Betriebssystems keine Dateien zerstört werden.

  Die meisten Texteditoren nennen die Datei vor dem Speichern zuerst um, damit bei einer Unterbrechung während des Schreibens man wenigstens noch die alte Version hat.

- Man sollte Benutzer vor potentiell zerstörerischen Aktionen warnen, und um eine Bestätigung bitten.

  Z.B. Editor verlassen ohne Speichern der Änderungen.

Effizienz (1)

- Was das Programm leisten muß, ist üblicherweise vorgeschrieben. Effizienz bedeutet dann möglichst geringen Resourcen-Verbrauch (Laufzeit, Speicher) zur Erreichung dieses Ziels.

  Wenn Programm A den gewünschten Wert in 1s berechnet, und Programm B den gleichen Wert in 30s, werden die meisten Benutzer A wählen. Dagegen würde ein Laufzeitunterschied von 1ms zu 100ms unwichtig sein. Oft hängt die Laufzeit aber an der Größe der Eingabe. Mit dem schnelleren Programm kann man häufig noch größere Eingaben in vernünftiger Zeit bearbeiten.

  Manchmal ist das Ziel nicht vollständig spezifiziert: Beispiel Schachprogramm: Pro Zug ist der Benutzer gewillt, 10s zu warten. Ein effizienteres Programm kann in dieser Zeit weiter vorausschauen, und mit der gegebenen Resource ein besseres Ergebnis erzielen.

Effizienz (2)

- Den größten Effekt auf die Effizienz hat der Algorithmus (das Berechnungsverfahren).

  Siehe Vorlesung über Algorithmen und Datenstrukturen.

- Z.B. wäre es günstiger, sich einen Wert zu merken, als ihn erst mit einer Schleife neu zu berechnen.
- Es gibt häufig einen Zielkonflikt ("Tradeoff") mit der Einfachheit.

  Z.B. ist es etwas schneller, über ein Array mit einem Pointer zu laufen, aber wenn man außerdem zählen muß, braucht man eine zusätzliche Variable.

Portabilität (1)

- Programme sollten möglichst unter Linux und unter Windows laufen, und auch auf weiteren Rechnern und Betriebssystemen.

  Natürlich muß das Programm auf der jeweiligen Plattform neu übersetzt (compiliert) werden.

- Man sollte also möglichst Funktionen der Standard-Bibliothek von C++ verwenden, und nicht direkte Betriebssystem-Aufrufe des jeweiligen Systems.

  Für graphische Benutzerschnittstellen ist das schwierig, aber es gibt Bibliotheken wie `Qt` oder `wxWidgets`, die für die verbreiteten Betriebssysteme erhältlich sind.
  Siehe auch [http://de.wikipedia.org/wiki/Liste_von_GUI-Bibliotheken].

Portabilität (2)

- Auch Eigenheiten ganz bestimmter Compiler sollten nicht ausgenutzt werden.
- Ebenso Eigenschaften der Hardware, z.B. hängt die Bytereihenfolge in einem `int` vom CPU-Typ ab.
- "Es gibt keine portabelen Programme, sondern nur Programme, die portiert wurden."

  Natürlich sollte ein Programm, das sich an den Sprachstandard hält, und nichts sonst verwendet, portabel sein. Aber man teste es besser schon während der Entwicklung gelegentlich, damit man nicht am Ende erst merkt, dass ein bestimmtes Konstrukt, was man sehr häufig verwendet hat, doch nur auf dem bestimmten Rechner läuft.

Portabilität (3)

- Wenn es sich nicht vermeiden läßt, systemspezifische Funktonen zu verwenden, beschränke man den systemspezifischen Anteil des Programmcodes auf möglichst kleine und gut-dokumentierte Teile.

  Man schreibt sich selbst dann eine system-unabhängige Funktion, die man im Rest des Programms verwendet. Die Implementierung dieser Funktion (ihr Rumpf) ist dann natürlich system-spezifisch. Man sollte Varianten für die wichtigsten Betriebssysteme (z.B. Linux/UNIX und Windows) schreiben. Mittels "bedingter Compilierung" kann man (automatisch oder halbautomatisch) die für das aktuelle System jeweils richtige Variante übersetzen lassen. Dies ist eine der Funktionen des C-Präprozessors, der später erläutert wird (mit den Befehlen `#ifdef` und `#if`).

Leichte Änderbarkeit (1)

- "Die einzigen Programme, die nie verändert werden, sind solche, die nie verwendet werden."
- Bei den meisten Programmen entdeckt man im Laufe der Zeit, dass es noch nützliche Erweiterungen geben würde.
- Programme dienen zur Lösung auf Aufgaben in der realen Welt, aber diese ist ständigen Änderungen unterworfen.

  Der Mehrwertsteuersatz ändert sich, die gesetzlichen Vorgaben werden an EU-Verordnungen angepasst, Firmen werden aufgekauft und in andere eingegliedert, u.s.w.

Leichte Änderbarkeit (2)

- Verständlichkeit des Programmcodes und eine gute Dokumentation für zukünftige Programmierer, die an dem Programm arbeiten sollen, sind wichtige Faktoren für die Änderbarkeit.
- Bei der Entwicklung des Programms sollte man auch darüber nachdenken, welche Änderungen bzw. Erweiterungen vorhersehbar / wahrscheinlich sind.

  Eventuell kann man den Programmcode ohne großen Zusatzaufwand so anlegen, dass diese Änderungen schon vorbereitet sind, und sich später leicht durchführen lassen.

Leichte Änderbarkeit (3)

- Dinge, die an einer Stelle im Programm konzentriert sind, sind meistens recht einfach zu ändern.

  Wenn man dagegen viele Stellen im Programm konsistent ändern muss, oder es viele undurchsichtige Zusammenhänge und Abhängigkeiten gibt, wird es schwierig.

- Z.B. sollten eher willkürliche Zahlwerte als Konstante (s.u.) definiert werden.

  Man kann die Konstante dann an vielen Stellen verwenden, aber ihr Wert ist nur an einer Stelle definiert. Eine Regel besagt, dass Programme nur die Zahlwerte 0 und 1 direkt im Programmcode enthalten sollten, alle anderen Zahlwerte nur in Konstantendeklarationen.

Leichte Änderbarkeit (4)

- Bei Klassen und Modulen sollte man von der "Kapselung" Gebrauch machen, d.h. möglichst wenig von außen zugreifbar machen.

  Wenn man dann die interne Implementierung ändern will, oder die Funktionalität erweitern, braucht man nur darauf achten, dass die bisher von außen zugreifbaren Funktionen noch das gleiche Verhalten wie vorher aufweisen. Die Schnittstelle, die von anderen Klassen/Modulen verwendet wird, sollte also möglichst klein sein.

- Man sollte nicht verschiedene Funktionalitäten in einer Klasse vermischen.

  Eine Änderung dieser Funktionalität würde sich dann durch Austausch der Klasse relativ leicht machen lassen.

Benutzerfreundlichkeit (1)

- Programme sollten sich möglichst bedienen lassen, ohne vorher eine lange Anleitung lesen zu müssen.

  "Wenn gar nichts anderes mehr hilft, lese man die Anleitung."

- Schlecht zu bedienende Programme können Benutzer unglücklich machen.

  Ich erinnere mich daran, wie an einer meiner früheren Unis die Sekretärin geweint hat, weil sie einfach die gewünschte Funktionen des Programms nicht gefunden hat, und der Chef schon darauf wartete. Es stellte sich heraus, dass das Programm die Menüs je nach Modus austauschte — nicht soviel, dass es sofort ins Auge sprang, aber diese Funktion stand eben in diesem Modus nicht zur Verfügung, und fehlte dann auch in dem Menü. Was ich mit diesem Beispiel sagen will, ist, dass Programmierer eine gewisse ethische Verantwortung haben.

Benutzerfreundlichkeit (2)

- Eine gute Online-Hilfe ist viel wert.

  Man sehe das nicht als lästige Aufgabe an, die man ganz zum Schluss erledigt, und die nur eine Alibi-Funktion hat.

- Software sollte sich möglichst intuitiv bedienen lassen, d.h. man folge etablierten Standards (was Benutzer von anderen Programmen gewöhnt sind).

  Ein überraschendes Verhalten des Programms sollte vermieden werden. Das Programm sollte sich möglichst konsistent verhalten.

- Der Benutzer sollte sehen können, was passiert.

  Fortschrittsanzeige. Aktionen sollten quittiert werden.

Benutzerfreundlichkeit (3)

- Fehlermeldungen sollten verständlich sein.

  Wie kann man das Problem lösen?

- Man beobachte Benutzer bei ihrem Umgang mit der Software (Usability-Tests).

  Man selber kennt das Programm ja und kann die Probleme, die andere Personen damit haben, nicht einschätzen.

- Vor gefährlichen Aktionen muss gewarnt werden.

  In 2005 hat ein Japanischer Akteinhändler (von Mizuho Securities) 610 000 Aktien von J-Com für 1 Yen pro Stück verkauft, anstelle der beabsichtigten einen Aktie für 610 000 Yen. Das waren mehr als 40 Mal so viele Aktien, wie es überhaupt gab. Die Firma mußte die nicht-existenten Aktien zurückkaufen und soll durch diesen Fehler 331 Millionen Dollar verloren haben.

Benutzerfreundlichkeit (4)

- Benutzerfreundlichkeit hat auch viel mit Anpassbarkeit an die Wünsche des Benutzers oder seine spezielle Hardwareumgebung zu tun:
  - ◇ Man Benutzer können nicht so gut sehen.

    Sie brauchen eine größere Schrift, oder kontrastreichere Farben.
  - ◇ Vielleicht hat der Benutzer einen sehr kleinen Bildschirm (PDA).

    Er möchte dennoch nicht viel scrollen, oder jedenfalls nur vertikal.
  - ◇ Manche Benutzer sind mit der Tastatur schneller und können sich viele Kürzel merken, andere kommen mit Maus und Menüs besser zurecht.

Inhalt

Konstanten (1)

Beispiel/Motivation:

- Oft ist es etwas willkürlich, wie lang man ein Array macht.
- Natürlich ist es absolut unverzichtbar, zu prüfen, daß längere Eingaben nicht akzeptiert werden.
- Aber eventuell möchte man später das Array vergrößern.
- Das wird schwierig, wenn die Array-Länge (z.B. 80) an mehreren/vielen Stellen im Programm steht.

Konstanten (2)

- Allgemein sollte man immer an spätere Änderungen des Programms denken, und jeden solchen "Parameter" nur an genau einer Stelle definieren.

  Sonst kann es sehr leicht passieren, daß man bei späteren Änderungen eine Stelle vergißt, was dann zu schwierig zu findenen Fehlern führt.
- Daher kann man mit

  ```
  const int max_input = 80;
  ```

  einen Namen für die Größe definieren, und im Rest des Programms nur über diesen Namen auf die maximale Anzahl Eingabezeichen zugreifen.

Konstanten (3)

- Das Array deklariert man dann mit:

```
char input[max_input+1];
```

(Das zusätzliche Zeichen wird für die Markierung '\0' des String-Endes benötigt.)

- Berechnungen nur mit Konstanten werden schon zur Compilezeit durchgeführt, so daß der Compiler die Array-Größe also kennt.
- Natürlich kann man den Wert einer Konstanten nicht mit einer Zuweisung etc. ändern.

Zeiger und Const (1)

- Man beachte, daß String-Konstanten, z.B. `"abc"` nicht geändert werden dürfen.

  Der Compiler kann sie in einem schreibgeschützten Bereich ablegen.

- Einen Zeiger in so eine Array-Konstante muß man deklarieren mit

```
const char *p;
```

- In diesem Fall ist nicht der Zeiger konstant, sondern `*p`, also der Bereich, auf den der Zeiger zeigt.
- Eine Zuweisung wie `*p = 'x';` wäre dann also verboten, nicht aber `p++`.

Zeiger und Const (2)

- Tatsächlich verhindert C++ im Moment noch nicht eine Zuweisung wie

```
char *q = "abc";
```

- Solche Zuweisungen gelten aber als "deprecated".

  D.h. "mißbilligt": Zukünftige Versionen von C++ können sie verbieten. In alten C-Versionen gab es kein `const`, und man wollte hier wohl die Kompatibilität wahren bzw. noch eine gewisse Schonfrist geben.

- Eine Änderung der Stringkonstante über `q` ist aber dennoch verboten:

```
*q = 'x'; // Fehler
```


Zeiger und Const (3)

- Der Compiler wird die unzulässige Zuweisung

```
*q = 'x'; // Fehler
```

nicht entdecken, weil `q` kein `const`-Zeiger ist.

- Es ist aber möglich, daß diese Zuweisung dann zur Laufzeit zu einem Fehler führt.

  Wenn der Compiler die String-Konstante in einem schreibgeschützten Speicherbereich abgelegt hat.

- Viele Compiler haben Optionen, um Stringkonstanten wie `"abc"` als konstante Arrays (`const char a[4]`) zu behandeln, so daß man nur mit `const char *p` darauf zeigen kann.

Zeiger und Const (4)

- Seien folgende Deklarationen gegeben:

```
const char *p1 = "abc"; // schreibgeschützt
char a[] = "abc"; // nicht schreibgeschützt
char * const p2 = a;
char *p3;
```

- `p1` ist ein änderbarer Zeiger auf einen konstanten Speicherbereich. Dagegen ist `p2` ein konstanter Zeiger auf einen änderbaren Speicherbereich.
- Die Zuweisung `p3 = p1` ist verboten, da dabei die `const`-Beschränkung wegfällt, also mit Zuweisungen über `p3` umgangen werden könnten.

Inhalt

Aufzählungstypen (1)

- Ein Aufzählungstyp ist ein Datentyp, der nur eine kleine Anzahl möglicher Werte hat, die durch explizite Aufzählung definiert werden.
- Die Werte sind Bezeichner (symbolische Konstanten), haben in C++ aber auch einen Integer-Wert.
- Beispiel:

```
enum weekdays {monday, tuesday, ..., sunday};
```

- Dieser Typ hat sieben mögliche Werte, nämlich die angegebenen Konstanten für die Wochentage Montag, Dienstag, ..., Sonntag (genauer: s.u.).

Aufzählungstypen (2)

- Wie von anderen Datentypen auch, kann man von dem Aufzählungstyp Variablen deklarieren, ihn als Ergebnis- oder Argumenttyp einer Funktion verwenden, oder als Komponententyp einer Struktur.
- Beispiel (Variablendeklaration):

```
weekdays w;
```

In C müßte man "`enum weekdays w;`" schreiben. Dies kann man sich dann mit einer `typedef`-Deklaration vereinfachen, s.u.

- Der Variablen `w` kann man dann einen der Werte zuweisen, z.B. `w = sunday;`

Aufzählungstypen (3)

- Sehr typisch ist auch ein Switch über allen möglichen Werten des Aufzählungstyps:

```
switch(w) {
case monday:
        cout << "Montag";
        break;
...
case sunday:
        cout << "Sonntag";
        break;
}
```

- Manche Compiler geben eine Warnung aus, wenn nicht alle Fälle behandelt sind.

Aufzählungstypen (4)

- Die Konstanten eines Aufzählungstyps haben einen Zahlwert, und werden in arithmetischen Ausdrücken implizit nach `int` umgewandelt.

  U.U. auch `unsigned int` oder `long`, falls die Zahlwerte dieses Typs zu groß für `int` sind.

- Normalerweise sind sie von 0 beginnend durchnumeriert:
  - ◇ `monday` hat den Wert 0.
  - ◇ ...
  - ◇ `sunday` hat den Wert 6.

Aufzählungstypen (5)

- Man kann aber auch explizit Zahlwerte festlegen:

```
enum weekdays {monday = 1, ..., sunday = 7};
```

  Man braucht nicht unbedingt für alle Konstanten des Aufzählungstyps einen Zahlwert festlegen: Es wird dann der Wert der letzten Konstante plus 1 verwendet. Im Beispiel würde man also den gleichen Effekt erreichen, wenn man nur für `monday` einen Wert festlegt.

- Während Aufzählungstypen automatisch nach `int` umgewandelt werden, gilt das Umgekehrte nicht.
- Bei Bedarf kann man folgende Notation verwenden:

```
w = weeksdays(3);
```

  In C muß man “`w = (weeksdays) 3;`” schreiben.

Aufzählungstypen (6)

- Ein häufiger Trick ist die Verwendung von Zweierpotenzen, um auch Mengen der Aufzählungswerte durch “Bit-oder” repräsentieren zu können:

```
enum weekdays {
        monday    =  1,
        tuesday   =  2,
        wednesday =  4,
        thursday  =  8,
        friday    = 16,
        saturday  = 32,
        sunday    = 64
}
```

Aufzählungstypen (7)

- Nun kann man z.B. mit `saturday|sunday` das Wochenende repräsentieren.

  In C++ gilt deswegen die Regel, daß der legale Wertebereich eines Aufzählungstyps (bei positiven Konstanten) sich immer bis $2^n - 1$ erstreckt, wobei n minimal so gewählt wird, daß alle Werte der Konstanten in den Bereich passen. Falls es negative Konstanten gibt, wird entsprechend ein Bereich der Form -2^n bis $+2^n - 1$ gewählt. Dies entspricht nicht der in anderen Sprachen üblichen Auffassung eines Aufzählungstyps (dort sind wirklich nur die angegebenen Konstanten legal), aber die zusätzlichen "anonymen" Werte kann man jedenfalls nur durch eine explizite Umwandlung einer Zahl in den Aufzählungstyp bekommen.

- Entsprechend kann man mit `w & sunday` testen, ob `w` den Sonntag enthält.

Inhalt

Typ-Deklarationen (1)

- Mit `typedef` kann man einen Namen für einen Typ deklarieren (ähnlich wie Variablendeklaration):

```
typedef int *int_ptr_t;
```

- Nun ist `int_ptr` im wesentlichen eine Abkürzung für `int *`.
- Man kann z.B. zwei Variablen deklarieren:

```
int_ptr x;
int *y;
```

  Die beiden Variablen `x` und `y` haben den gleichen Typ (können einander zugewiesen werden).

Typ-Deklarationen (2)

- Weil `typedef` keinen neuen Typ einführt, sondern nur einen neuen Namen für einen existierenden Typ, erhöht es nicht direkt die Typ-Sicherheit des Programms.

  Der Compiler findet nicht mehr Fehler. Allerdings ist der Programmtext eventuell für den Programmierer verständlicher, so dass dadurch weniger Fehler auftreten. Übrigens gibt es auch Programmiersprachen, in denen die Typ-Deklaration einen echt neuen Typ einführt. In C/C++ muss man dafür `struct` oder `class` verwenden (siehe Kapitel 11). Auch eine Struktur mit einer einzigen Komponente ist ein ganz neuer Typ, und nicht mehr zuweisungskompatibel zum Typ der Komponente.

Typ-Deklarationen (3)

- `typedef` wird meist verwendet, um einen Zeigertyp für Klassen einzuführen.

  Da die Objekte oft relativ groß sind, arbeitet man im Programm meist mit Zeigern auf die Objekte. Das ist z.B. auch wichtig, wenn ein Objekt einen änderbaren Zustand hat: Bei einer Zuweisung auf Objekt-Ebene würde das Objekt normalerweise kopiert.

- Es wäre recht mühsam, wenn man bei Variablen jeweils den “*” explizit schreiben muss, durch einen neuen Typ kann man das verstecken.

  In C war `typedef` noch wichtiger, weil da `struct/enum` T `{...};` einen Typ definiert, den man immer mit `struct` T bzw. `enum` T ansprechen musste. In C++ definieren diese Konstrukte direkt einen Typ T.

Inhalt

1. Grundsätze guten Programmierstils
2. Konstanten
3. Aufzählungstypen
4. Typdeklarationen (`typedef`)
5. Zusicherungen (`assert`)
6. Warnungen des Compilers

Zusicherungen (1)

- Mit `assert` kann man Bedingungen, von denen man annimmt, dass Sie an einer bestimmten Stelle im Programm immer erfüllt sein müssen, prüfen lassen.

  Man verspricht gewissermassen, dass die Bedingung wahr ist. Deswegen nennt man es “Zusicherung” (engl. “Assertion”).

- Sollte die Bedingung nicht erfüllt sein, wird das Programm mit einer Fehlermeldung abgebrochen.
- In der Fehlermeldung steht auch die Quelldatei und Zeilennummer des Aufrufs von `assert`, bei neueren Versionen auch der Name der aktuellen Funktion.

Zusicherungen (2)

- Um `assert` benutzen zu können, muß man die entsprechende Header-Datei einbinden:

```
#include <cassert>
```

- Dann wird `assert` wie eine Funktion mit booleschem Parameter benutzt:

```
assert(n > 0);
```

  Es ist keine Funktion, sondern ein “Macro” (siehe Kapitel über den C-Präprozessor). Nur so ist es möglich, dass `assert` im Fehlerfall den Programmtext der Bedingung, sowie die Position im Quellprogramm ausgibt. Eine echte Funktion “weiss” nicht, von wo sie aufgerufen wurde und bekommt das Argument nur in ausgewerteter Form.

Zusicherungen (3)

- Ist im obigen Beispiel `n` nicht größer als 0, bekommt man folgende Meldung:

```
test: test.cpp:10: int main():
Assertion ‘n > 0’ failed.
Aborted
```

Das Programm wird dann abgebrochen.

Das Abbrechen geschieht über die Funktion `abort()`, die normalerweise eine Datei “`core`” schreibt, die ein Abbild des Hauptspeichers dieses Prozesses zum Zeitpunkt des Programmabbruchs ist. Mit einem Debugger wie `gdb` kann man sich dann z.B. Variableninhalte zum Zeitpunkt des Fehlers anschauen. Sollte `core` nicht geschrieben werden, versuche man “`ulimit -c unlimited`”. Eventuell enthalten Dateien wie `.bashrc` eine Beschränkung, die man zuerst löschen muss (und sich wieder einloggen), notfalls braucht man die Hilfe eine Administrators.

Zusicherungen (4)

- Vorteile der Verwendung von `assert`:
  - ◇ Der Fehler wird näher an der eigentlichen Fehlerursache bemerkt.

    Ohne `assert` läuft das Programm häufig noch ein Stück und stürzt dann ab, oder produziert am Ende einfach einen falschen Wert. Je näher man an der Fehlerursache ist, desto einfacher wird es, den Fehler zu finden.
  - ◇ Der Leser des Programms bekommt eine wichtige Zusatzinformation.

    Es ist ähnlich wie ein Kommentar, aber da es überprüft wird, kann man sich darauf eher verlassen.

Zusicherungen (5)

- Vorteile der Verwendung von `assert`, Forts.:
  - ◇ `assert` ist Teil der defensiven Programmierung: Man schützt sich vor fehlerhaften Aufrufen.

    Wenn verschiedene Module/Klassen von verschiedenen Programmierern entwickelt werden, oder ein Modul/eine Klasse später in einem anderen Kontext wieder verwendet wird, kann es leicht zu Missverständnissen kommen: Mit `assert` können sie schneller aufgeklärt werden.
  - ◇ Falsche Annahmen fallen schneller auf.

    Wenn eine Zusicherung fehlschlägt, bedeutet das nicht immer, dass das Programm vor der Zusicherung wirklich einen Fehler enthält: Nicht selten ist auch die Zusicherung falsch. Auch solche falschen Annahmen müssen aber aufgeklärt werden.

Zusicherungen (6)

- Da die Tests Laufzeit kosten, kann man sie abstellen, wenn man glaubt, dass das Programm keine Fehler mehr enthält.

  Es wird gesagt, dass das so ist, als würde man nur in der Fahrschule mit Gurt fahren, aber später ohne. Wenn der Benutzer die Fehlermeldung auch nicht verstehen kann, so wüsste er wenigstens, dass etwas nicht in Ordnung ist, und er dem Ergebnis nicht vertrauen kann.
- Dazu wird das Programm mit der Option `-DNDEBUG` compiliert.

  Das Macro `NDEBUG` (“no debugging”) muss definiert sein, wenn die Datei `assert.h` vom Compiler (Präprozessor) verarbeitet wird. Der Präprozessor-Befehl `#define NDEBUG` hätte den gleichen Effekt.

Zusicherungen (7)

- `assert` ist ausschließlich zur Entdeckung von Programmierfehlern gedacht, nicht für Fehler, die auch bei korrektem Programm auftreten können, z.B.:
  - ◇ falsche Benutzereingaben,
  - ◇ kein Speicher mehr (`new`/`malloc` schlagen fehl),
  - ◇ Ausgabedatei existiert und ist schreibgeschützt.
- Der Grund ist, dass man die Prüfung von Assertions abstellen kann (s.o.), aber obige Fehler immer behandelt werden müssen.

  Außerdem würde es den Leser des Programms verwirren, wenn `assert` falsch verwendet wird.

Inhalt

Warnungen des Compilers (1)

- Es gibt Fehler,
  - ◇ die durch Testen relativ schwierig zu finden sind,
  - ◇ aber ganz leicht, wenn man eine passende Warnung des Compilers anschaltet.
- Es empfiehlt sich also, die meisten Warnungen immer angeschaltet zu haben.
- Der Compiler (und spezielle Werkzeuge wie `lint`) führen eine statische Analyse des Quelltextes durch.

  Statisch, weil ohne das Programm auszuführen.

Warnungen des Compilers (2)

- Z.B. kann man mit einer einfachen Heuristik versuchen, nicht initialisierte Variablen zu finden:
  - ◇ Gibt es einen Ausführungspfad durch eine Prozedur, bei dem man eine Verwendung der Variablen erreicht, ohne vorher an einer Zuweisung vorbei gekommen zu sein, wird eine Warnung generiert.
- Das ist nur eine Heuristik. Es kann nicht immer funktionieren, weil man beweisen kann, dass das Problem unentscheidbar ist.

Warnungen des Compilers (3)

- Die obige Heuristik produziert sowohl
  - ◇ falsch positive Ergebnisse (eine Warnung wird angezeigt, obwohl gar kein Fehler vorliegt),

    Manche Ausführungspfade, die bei der statischen Analyse betrachtet werden, sind gar nicht möglich, weil die Bedingungen untereinander abhängen, z.B. wenn die Bedingung vom ersten `if` (mit Zuweisung) falsch war, dann muß auch die Bedingung vom zweiten `if` (mit Verwendung) falsch sein. Solche Abhängigkeiten betrachtet der Compiler im allgemeinen nicht.
  - ◇ als auch falsch negative Ergebnisse.

    Man bekommt keine Warnung, obwohl man auf eine nicht initialisierte Variable zugreift. Z.B. könnte ein Array nur teilweise initialisiert sein. Die Analyse für globale Variablen ist auch zu kompliziert.

Warnungen des Compilers (4)

- Dennoch sind die Warnungen nützlich: Viele Fehler werden so schnell gefunden.
- Wenn man eine Warnung bekommt, und sicher ist, dass das Programm korrekt ist, versuche man dennoch, es so umzuformulieren, dass die Warnung nicht mehr ausgegeben wird.

  Wenn man nichts gegen die fehlerhaften Warnungen unternimmt, werden es im Laufe der Programmentwicklung immer mehr. Dann fällt die eine wichtige Warnung unter den vielen, die man schon im Kopf "abgehakt" hat, nicht mehr auf. Viele Compiler haben speziell formatierte Kommentare oder `#pragma`-Anweisungen, mit denen man falsche Warnungen abstellen kann.

Warnungen im g++ (1)

Empfohlene Warnungen:

- `-Wall`

  Entgegen dem Namen schaltet `-Wall` nicht alle Warnungen ein, sondern nur diejenigen, bei denen sich die `g++`-Programmierer relativ sicher waren, dass sie wirklich auf einen Fehler im Programm hinweisen. Gelegentlich wird eine Warnung ausgegeben, obwohl der Programmcode korrekt und vom Programmierer so beabsichtigt ist.
- `-Wextra`

  Dies sind einige zusätzliche Warnungen, aber noch immer nicht alle Warnungen, die der `g++` produzieren könnte.

Warnungen im g++ (2)

Empfohlene Warnungen, Forts.:

- `-Wshadow`
- `-Wpointer-arith`
- `-Wcast-qual`
- `-Wcast-align`
- `-Wstrict-prototype`
- `-Wunreachable-code`
- Siehe auch die Liste im Handbuch.

  [http://gcc.gnu.org/onlinedocs/gcc-4.5.1/gcc/Warning-Options.html]

Funktionsattribute (1)

- Viele Funktionen geben einen Wert zurück.
- Es ist normalerweise falsch, wenn dieser Wert einfach ignoriert wird.

  Entweder ist der Funktionsaufruf sinnlos (wenn der einzige Zweck der Funktion ist, den Wert zu berechnen), oder Sie ignorieren eine Fehlermöglichkeit. Leider kommt das so häufig vor (und Fehlerw erden inzwischen mit Exceptions besser behandelt), dass dieser Test nicht Standard ist.
- Schreiben Sie z.B. `sqrt(2);` statt

  ```
  x = sqrt(2);
  ```

  so gibt es eine Warnung.

Funktionsattribute (2)

- Schreiben Sie hingegen

  ```
  double approxsqrt (double x) {
    return (1+x)/2;
  }
  int main () {
    approxsqrt(2);
    return 0;
  }
  ```

  so gibt es keine Warnung.
- Abhilfe: Deklarieren Sie

  ```
  double __attribute__((warn_unused_result))
    approxsqrt (double x)
  ```


Funktionsattribute (3)

- Die Notation mit `__attribute__` ist `g++`-spezifisch.
- Man kann sie aber leicht mit dem Precompiler eliminieren lassen.
- Dazu muss man den Befehl

  ```
  #define __attribute__(X)
  ```

  (auf einer eigenen Zeile) in der Version für Nicht-`g++` Compiler ausführen lassen.

  Das bekommt man auch mit dem Precompiler hin, z.B. schliesst man diese Anweisung in “`#ifndef __GNUC__`” und “`#endif`” (jeweils auf eigenen Zeilen) ein.