PraxProg Programmierkonventionen

24. November 1998

1 Präambel

Dies ist eine kommentierte Programmierkonvention für den Kurs ,,Praxis des Programmierens" im Wintersemester 98/99.

§ 1 Diese Programmierkonventionen haben immer recht.

§ 2 Sollten die Programmierkonventionen einmal nicht recht haben, so tritt automatisch Paragraph 1 außer Kraft. (Beispiel: wiederverwendeter Code.)

2 Grundlagen

§ 3 Programmiersprache ist ISO-C++. Insbesondere sind C-Relikte wie

- `printf`
- `struct`
- `malloc` und `free`
- Durch `inline`, `template` und `const` vermeidbare `#defines`

zu vermeiden. Da leider noch nicht alle Compiler den vollen Sprachumfang beherrschen, werden wir allerdings nur konservativen Gebrauch von neueren Features machen. Folgende Eigenschaften werden *nicht* empfohlen:

- Namespaces
- Ausnahmebehandlung
- Mehrfachvererbung
- Komplizierte Kombinationen von `templates` und Overloading.

§ 4 Alle Bezeichner werden in englisch verfaßt, alle Kommentare und Meldungen des Programms in deutsch.

2.1 Bezeichner

§ 5 Keine Underscores (_). [1] Ausnahme: Wenn Libraries (LEDA) dies erfordern, oder die Analogie zu einer LEDA-Funktion deutlich gemacht werden soll.

§ 6 Konstanten und Makros werden ausschließlich mit Großbuchstaben bezeichnet. Außerdem: globale Konstanten werden vermieden, statt `#define FORTYTWO 42`,

```
class PlanEditor {
  static const int FORTYTWO;
};
```

in der Implementation dann:

```
const int PlanEditor::FORTYTWO = 42;
```

§ 7 Ansonsten bestehen Bezeichner aus Kleinbuchstaben, mit folgenden Ausnahmen:

- Klassenbezeichner und Typbezeichner fangen mit Großbuchstaben an.
- Bei zusammengesezten Worten ist der erste Buchstabe eines neuen Wortes groß (z.B. `E fooBar(E a);`).
- Makros, fangen mit einem Großbuchstaben an und werden nur verwendet, wenn sie nicht durch ein C++ Konstrukt ersetzt werden können.

§ 8 Abkürzungen sind soweit als möglich zu vermeiden.[2] Sollten sie doch irgendwann einmal nötig werden, so wird die folgende Vorgehensweise empfohlen:

1. Suche nach kürzeren Wörtern, die genauso bezeichnend sind.
2. Suche nach gängigen Abkürzungen für Wörter.
3. Wenn eine Abkürzung verwendet wird sollte sie überall gleich verwendet werden. Beispiel `Constr` für `Constraint`.

---

[1] Darüber kann man wunderbar debattieren. Freunde des „_“ behaupten er erhöhe die Lesbarkeit. Wir halten dagegen:

1. „_„ zerstört optisch die Tatsache, daß es sich bei einem Bezeichner um eine lexikalische und semantische Einheit handelt. Hinter den optisch sehr ähnlichen Operatoren „`.`“, „`->`“, „`-`“ etc. versteckt sich Code hinter „_„ nicht.
2. Die meisten Autoren schrecken vor mehreren „_„ in mehrfachen Zusammensetzungen zurück. Dadurch wird es fast unmöglich sich die Schreibweise von Bezeichnern zu merken.
3. „_„ kostet Platz, den man besser für echte Information investiert (z.B. längere Bezeichner).
4. Zusammensetzungen können genauso effektiv durch die unten gegebenen Regeln lesbar gemacht werden. (Das Deutsche kommt sogar ohne aus.)

[2] Selbst der Autor kann sich oft nicht merken wie was abgekürzt wurde. Und ein Außenstehnder hat es natürlch noch schwerer.

4. Benutzung der ersten ein bis zwei Silben eines jeden Teilwortes.
5. Weglassen der Vokale. Dabei sollten aber max. 4–5 Buchstaben pro Teilwort übrigbleiben. (Zum Beispiel wäre `Cnstrnt` ein wahrer Zungenbrecher.)
6. Kombination von 4. und 5.

§ 9 Verwandte Klassen sollten verwandte Namen haben. Beispiel `Sender`, `AmateurSender` oder `PlanTabularView` und `PlanTabularContr(oller)`.

2.2 Formatierung

2.2.1 White Space

§ 10 Keine `TAB`-Zeichen. Einrückungen bestehen aus Leerzeichen und die Standardeinrückung ist 2 Leerzeichen.

§ 11 Mindestens eine (besser zwei) Leerzeilen vor jeder Funktionsdefinition.

§ 12 Eine Leerzeile zum Trennen von logischen Codeblöcken innerhalb einer Funktion.

§ 13 Binäre Operationen außer „`.`“ und „`->`“ werden in der Regel von Leerzeichen umgeben. Ausnahme: Die Leerzeichen können entfallen wenn Prioritäten angedeutet werden sollen. Zum Beispiel `x = 2*a + b;`.

§ 14 Zwei Leerzeichen nach dem Semikolon in `for`-Schleifen:

```
for (i = 0;  i < n;  i++) {...}
```

§ 15 Ein Leerzeichen vor und nach einer Bedingung für die Kontrollstrukturen `if`, `for`, `while`. Aber *kein* Leerzeichen beim Prozeduraufruf `x = foo(2);`.

2.2.2 Klammerung

§ 16 Öffnende Klammern für Kontrollstrukturen auf der gleichen Zeile wie das Schlüsselwort. Bei Funktionen aber in Spalte 1 der folgenden Zeile.

§ 17 Alle Kontrollstrukturen sind zu klammern.[3] Ausnahme: else-if Ketten.

Beispiel:

[3] Das vermeidet eine Fehlerquelle beim Einfügen von zusätzlichen Operationen. Insbesondere Debug-Hilfen werden oft sehr eilig eingefügt.

```
// bar knodeln Variante 17: quer verschraenkt
// Pre: Vollmond
// Post: Bar ist querseitig geknodelt
void Bars::knodel(void)
{ int i;

  for (i = 0;  i < 10;  i++) {                                       // Bla
    if (i = 3) { bar(i); }
    else if (fooBar(i) < 42) {
      if (i = 7) { bar(i + 1);} else {bar(i - 1); }
      else {                                             // geniale Enthuellung
        cout << "no" << endl;

        // Kommentar vor logischem Block
        bar(3);
        bar(4);

        // noch'n Block
        bar(5);
      }
  }
}
```

§ 18 Lieber 10 Klammern zu viel, als eine zu wenig z.B. `2 + ((x & 3) << 4)`. Nicht jeder Programmierer beherrscht die 42 Klammerebenen von C im Schlaf.

2.3 Kommentare

§ 19 Der alte C-Kommentar `/* ... */` wird nur zum Auskommentieren von Code-Blöcken während der Programmentwicklung eingesetzt.

§ 20 Kommentare sollen den Quelltext nicht paraphrasieren sondern ergänzen. Der beste Kommentar ist einer, der gar nicht da ist, weil die Programmierung so kristallklar ist.

§ 21 Anfang einer einzelnen Kommentarzeile ist ein Leerzeichen z.B. `// this is a comment`.

§ 22 Kommentarzeilen innerhalb einer Funktion werden mit dem Code eingerückt und stehen oft nach einer Leerzeile, die einen logischen Block abtrennt.

§ 23 Kommentare innerhalb einer Zeile, in der auch code steht sind innerhalb einer Funktion linksbündig oder rechtsbündig ausgerichtet.[4]

§ 24 Programmstellen, die noch zu ändern sind, in Verdacht stehen, einen Fehler zu enthalten oder sonstwie markiert werden sollen, werden mit einem Kommentar versehen, der mit `//?` beginnt.

[4] Linksbündig sieht besser aus und ist leichter wartbar. Oft ist dafür aber nicht genug Platz.

§ 25 Hacks müssen immer kommentiert werden und beginnen mit `//>^`. (Die Zeichenkonvention soll eine gefährliche Kurve symbolisieren.)

§ 26 Vor jeder Funktionsdeklaration steht ein Kommentarblock:

1. Informelle Beschreibung der Funktion
2. Formale Vorbedingungen (eingeleitet durch `Pre:`)
3. Formale Nachbedingungen, d.h., Zusicherungen, die die Funktion macht.

Formale Vor- und Nachbedingungen, die in lesbarer Form als `assert` geschrieben werden können, braucht nur in dieser Form geschrieben werden (Vorbedingungen am Anfang der Funktion, Nachbedingungen vor dem `return`). Der Kommentar am Anfang kann entfallen, wenn die Dokumentation länger würde als der Code selbst. Beispiel `double getRange()` { `return range;` } Vor einer Funktionsdefinition, die woanders deklariert ist steht nur ein Kommentar, der Implementierungsbesonderheiten erläutert.

§ 27 Eine Datei wird durch Zeilen der Form

```
//============================ bla bla bla ================================
//---------------------------- bla bla bla --------------------------------
```

In Abschnitte respektive Unterabschnitte unterteilt.

§ 28 Vor jeder Klassendefinition steht ein Kommenarblock mit folgenden Komponenten:

1. Klartextbeschreibung des Datentypen einschließlich Änderungen gegenüber Basisklassen etc. Kurz fassen! Lieber Übersicht und Verweise auf weitere Kommentare an der relevanten Stelle.
2. Klasseninvarianten, die mehrere members gleichzeitig betreffen (Block beginnt mit `//Inv:`

§ 29 Anforderungen an einzelne members werden duch Kommentare der Form `Inv:` erklärt. Fehlender Kommentar bedeuted, daß alle Werte des angegebnene Typen erlaubt sind. Beispiel: `double range; // Reichweite //Inv: >= 0`

§ 30 Klassen, die von mehreren Teilprojekten benutzt werden sind mit den LEDA üblichen Kommentaren zu versehen, damit mit `lweb` eine Dokumentation erstellt werden kann. (Aufgabe der Integrationsgruppe). (Siehe auch Abschnitt 1.2. im LEDA-Handbuch.

§ 31 Jede Headerdatei beginnt mit Kommentarzeilen folgenden Inhalts:

1. Ursprünglicher Autor (+ Teilprojekt)
2. Einigen Zeilen Funktionsbeschreibung

3. Jeweils eine Zeile für Änderungen. Mindestens muss hier jede Revision angegeben sein, die an andere Teilprojekte weitergegeben wurde. Diese besteht aus:
   - Datum
   - Gegenstand der Änderung. Beispiel: „Doppelt querseitiges knodeln statt einfachem quibbeln".
   - Wer hat die Änderung vorgenommen

Alternativ kann das obige Change-Log auch mit RCS geführt werden (versuchen Sie mal `man rcsintro`.

§ 32 In Code-Dateien kann die Funktionsbeschreibung entfallen. Dafür ist dort allgemeines zum Implementierungsansatz zu sagen. Das Change-log beschränkt sich auf Änderungen, die nicht ebenfalls die Header-Datei betreffen.

2.4 Aufbau einer Klassendefinition

§ 33 Reihenfolge der verschiedenen Teile einer Klasse (falls vorhanden):

1. `friend`s
2. Public Members
3. Public `static` Members
4. Public Konstruktoren
5. Public Destruktor
6. Public Zuweisungsoperatoren
7. Public Member-Funktionen
8. Private Members
9. Private `static` Members
10. Private Member-Funktionen

3 Dateien

§ 34 Jedes Modul hat eine eigene Headerdatei (`.h`). Diese hat folgenden Aufbau:

```
#ifndef <MODULNAME>
#define <MODULNAME>
// Kommentare wie oben erklaert
...
#endif
```

In der Regel besteht ein Modul aus einer Klasse oder mehreren nahe verwandten Klasse. Der Modulname ist oft der Name der wichtigsten Klasse im Modul (besteht aber nur aus GROSSBUCHSTABEN weil er ein Makro ist.) Der Dateiname ist ebenfalls der gleiche Name (aber nicht alles grossgeschrieben.) Im Headerfile sind auch alle inline-Funktionen zu definieren.

§ 35 Non-inline Funktionen sind in einer Datei mit der Endung `.C` unterzubringen. Ansonsten ist der Dateinname der gleiche wie für das Headerfile. Module ohne `.C` File, die ausschließlich inline Funktionen enthalten, sind erlaubt und erwünscht. Diese sollten aber:

- Nicht zu viel Code
- Keine häufig verwendeten längeren Funktionen

enthalten.[5]

4 Was (bisher) nicht geregelt ist

In vielen Projekten gibt es noch detailliertere Konventionen.

- Detailliertere Kommentar- und Dokumentationsregeln.
- Die Anfangsbuchstaben eines Variablenbezeichners codieren den Typ. (Bei objektorientierten Projekten klingt das schwierig.)
- Festlegung der Dateihierarchie für die Module
- Benutzungsregeln für Bibliotheken. Beispiel: Nach einem `new` immer ein Test ob Allozierung erfolgreich war.

[5]Beides erhöht Übersetzungszeiten und Code-Größe.