Programming Guidelines für C++

Max Kaluschke

October 6, 2014

- Basiert auf http://cgvr.informatik.uni-bremen.de/progr/guidelines.shtml

Ziele

- Gemeinsamer Schreibstil
  - einfacher anderen Code verstehen
  - keine Diskussion um Stil
  - verhindert einfache Fehler
- Auch auf einen semantischen Stil einigen
  - gutes Design erspart Bugsuche

10 Gebote

1. Ich will niemals hören: “Ich weiß, daß man X noch machen müßte, aber das mache ich später, wenn alles läuft”
2. Nur eleganter Code ist guter Code.
3. Kommentiere! (Es steht zwar im Prinzip im Code, aber keiner hat Lust auf Reverse-Engineering!)
4. Wenn Du die Regeln verletzen mußt, kommentiere warum (und nicht daß Du sie verletzt hast).
5. Verwende bezeichnende Namen (“labeling names”) und eine einheitliche Namenskonvention.
   - Wähle Namen von Funktionen, Variablen etc sorgfältig!
6. Achte auf übersichtliches Indenting und Spacing!
7. Frage Dich beim Schreiben immer “was ist wenn . . . ”! (vollständige Fallunterscheidung)
8. Schreibe nie Code mit Nebeneffekten!
9. Wenn es doch sein muß, kommentiere diese ausführlich und unübersehbar!
10. Lerne Deine Tools vollständig zu beherrschen.

Namenskonventionen

- Namenswahl extrem wichtig
  - schlechte Namen reichen aus Code unlesbar zu machen
  - besonders bei OO-Design
  - zu lange Klassennamen deuten zB auf Designfehler hin

Namenskonventionen

Art	Beispiel	Erläuterung
lokale Variablen	mylocalvar, my_local_var	alles klein
Instanzvariablen	m_var	m steht fuer “member”
Klassennamen	MyCoolClass	Anfang groß
Exception-Klassen	XThrewUp	So wie Klassennamen, aber beginnend mit ‘X’
Konstanten	const int MaxNum = 10;	ähnlich wie Klassenvariablen
Methoden	calcSomeCoolValue()	inCaps, klein fangen (ausführlicher)
Template-Parameter	<MyTypeT>	wie Klassennamen mit T am Ende
abfragende Funktion	getVar(..) const	fragt Variable _var ab

Namenskonventionen

Art	Beispiel	Erläuterung
modifizierende Funktion	setVar(..)	
Eigenschaft	isProperty()/hasProperty() const	
typedef	SomeThingT	analog zu Klassennamen, mit 'T'
Defines	#define DEBUG_ON	all-caps mit Underscores
enum-Typen	myEnumTypeE	Suffix E
enum-Member	COL_TRUE, COL_FALSE	wie Defines
Pointer- oder Reference-Deklarationen	String *str, &str2;	* und & steht bei der Variable, nicht beim Typ

Kommentare

- nur so viel wie nötig
- Während des Programmierens kommentieren!
  - Da sind die Gedanken noch frisch
- Verschiedene Arten von Kommentaren:
  1. Am Anfang Klassen
     - "big picture" Erklärung
     - Besonderheit (zB Compile-Falgs)
  2. Vor Funktionen
     - Beschreibung
     - Parameter (besonders Ausgabe Parameter kennzeichnen)
     - Pre-/Postconditions
     - Seiteneffekte
  3. Vor Variablen
     - immer vor globalen Variablen!
  4. Zwischen Codeabschnitten
     - beschreiben **was** gemacht wird, nicht **wie**

Kommentare

- komplette Klassentemplates mit Kommentarbeispielen auf http://cgvr.informatik.uni-bremen.de/progr/template-comments.cpp
  - und viele andere Beispiele!

Layout

- Pro Zeile
  - 80 Zeichen
  - Ein Statement
- Klassenlayout
  - public
    1. Konstanten
    2. Typen
    3. Variablen
    4. Methoden
  - protected
    - -“-
  - private
    - -“-

Layout

Headers

- sollte nur für Anwender interessanten Inhalt enthalten
  - keine unnötigen Implementierungsdetails
- per `#ifndef` gegen Dopellinklusion schützen

```
#ifndef _CLASS_NAME
#define _CLASS_NAME

// Tatsächlicher Inhalt des Headers

#endif
```

- Man sollte in jeder .cpp Datei die entsprechende .h Datei includen
  - Dadurch prueft der Kompiler, dass die Signaturen uebereinstimmen

Layout

Einrűckung

- If-Then-Else

```
for ( ... )
{
    if ( .. )
    {
        ..
    }
    else
    {
        ...
    }
}
```

Layout - Einrűckung

Was wir **nicht** wollen:

```
for ( ... ) {
    if ( .. ) {
        ..
    } else {
        ...
    }
}
```

Layout - Einrűckung

- Variablen (wenn mőglich):

```
int                 x, y;                   // Beschreibung
int                 xturns, yturns;         // Beschreibung
objPolyhedronP      p1, p2;                 // Beschreibung
int                 i;                      // Beschreibung
```

Layout - Einrűckung

- Whitespaces in Bedingungen
- Was wir **nicht** wollen:

```
for(i=o->begin();i<o->l()&&o->M(i)!=-1;i++){
    o->M(i)=-1;
}
```

Layout - Einrűckung

- **besser**:

```
for( i = o->begin(); i < o->l() && o->f(i) != -1; i++ )
{
    o->f(i) = -1;
}
```

Wartbarkeit

- Faktorisieren
  - zB: bar() implementiert selben Inhalt wie foo() mit einem Zusatz
    - foo() innerhalb von bar() aufrufen
    - so kann der Inhalt nur an einer Stelle geändert werden
  - genauso innerhalb von Funktionen, zB bei if-then-else oder switch-case
- Genrell Code **vermeiden**, der leicht missverstanden werden kann
  - zB **keine** Zuweisung oder Statements als Ausdruck

```
if(b = b2)
if(i++ < c)
```

- nur wenn es den Code lesbarer macht

Generelles

- Warnings beim Compilen anschalten und entsprechend Code schreiben der keine Warnungen erzeugt
- asserts verwenden damit Bugs möglichst früh erkannt werden
- Namespaces verwenden!
  - niemals in Headern `using namespace` aufrufen
- Rufe im Konstruktor immer den Konstruktor der Basisklasse auf
- Überschreibe **nicht** geerbte Default-Parameter

Generelles

Casts

- Moderne cast Varianten verwenden
  - zB static_cast(obj) statt (T)obj
  - Grund:
    - Korrektheit: es gibt verschiedene Arten von Casts die bei der alten Methode nicht unterschieden werden
    - Einfacher zu finden da es schwer ist nach (...) zu suchen

Generelles

Exceptions

- Objekte sollten auch bei geworfenen Exceptions nicht in undefinierten Zustand gelangen oder Speicherleks entstehen
  - zB keine Exceptions in Dekonstruktoren werfen
  - Aufpassen wenn man *new* im Konstruktor verwendet, da bei einer Exception der Dekonstruktor nicht aufgerufen wird
    - $\rightarrow$ *try* verwenden

Generelles

Methoden, Funktionen

- **virtual** Methoden, sollten innerhalb der Vererbungshierarchie ueberall so deklariert werden
  - verdeutlicht, dass sie ueberladen werden kann
- **const** immer verwenden wenn die aktuelle Instanz nicht durch die Methode veraendert wird
- call-by-value bei Objekt-Parametern vermeiden
- Parameter auf Gueltigkeit pruefen

Generelles

Pointer vs. Reference

- Wenn Parameter als Reference deklariert wird, sollte man diesen als *const* deklarieren
  - **Konvention**: Pointer dürfen innerhalb einer Funktion veraendert werden, Referenzen aber nicht

Generelles

Verschiedenes

- "Data hiding": getter und setter verwenden, statt Attribute *public* zu deklarieren
- keine Initialisierung von Instanzen per Zuweisung!
  - Konstruktor wird uU zweimal aufgerufen

```
A a = A();        // verboten
```

- besser:

```
A a();
```

Generelles - Verschiedenes

Floats

- Aufpassen wenn man mit *float* s arbeitet
  - Falsch:

```
if ( x == a )
    ...
if ( x < a )
    ...
if ( x > a )
    ...
```

* Richtig:

```
if ( fabs(x - a) <= epsilon * fabs(a) )
```

Generelles - Verschiedenes

- Hardgecoded Pfade vermeiden
  - stattdessen relative Pfade mit entsprechender Fehlerpruefung
- “Magic numbers” vermeiden

```
void foo( int bla )
{
    if ( bla == 1 )
        ..
    else
    if ( bla == 2 )
        ..
}
```

Generelles - Verschiedenes

- enums sind eine bessere Loesung:

```
typedef enum
{
    Fall1, Fall2, ...
} FooFaelleE;

void foo( FooFaelleE bla )
{
    ...
}
```

Generelles - Verschiedenes

Includes

- Standard Includes

```
#include <...>
```

- projektspezifische Includes

```
#include "..."
```

Generelles - Verschiedenes

malloc

- Es macht Sinn einen malloc-Wrapper in folgender Form zu benutzen

```
#define xmalloc( PTR, SIZE, ACTION )                    \
{                                                       \
    PTR = malloc( SIZE );                               \
    if ( ! PTR )                                        \
        ACTION;                                         \
}
```

- dadurch ist man gezwungen eine Aktion fuer den Fehlerfall vorzusehen