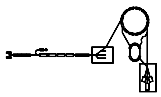

C/C++ Styleguide

U. Krause

GSI, Gesellschaft für Schwerionenforschung mbH
Planckstraße 1, D-64291 Darmstadt
Tel.: 06159/71-0

Issued: 18. Dez. 2000
Key: O-SIS-10
Printed: 5. März 2012

Diese Notiz stellt Empfehlungen für das Abfassen von Programmen in den Programmiersprachen „C“ und „C++“ zusammen.

Änderungsprotokoll		
Datum	Name	Kommentar
18. 12. 2000	UK	Erste Version
09. 05. 2001	UK	Pointerschreibweisen, Index
19. 11. 2004	LH	Enums sind Konstanten; Typwandlung in C++
6. 04. 2011	LH	Includes for C++ added

Inhaltsverzeichnis

1 Einleitung

1.1 Zielsetzung

Die Programmiersprache „C“ ist sehr mächtig in ihren Möglichkeiten. Leider bietet die Sprachsyntax nicht unbedingt die Klarheit, die von anderen Programmiersprachen nahegelegt wird. Beides führt dazu, dass sich komplexe Funktionalitäten zwar sehr knapp, aber auch sehr schwer verständlich formulieren lassen.

Um sich schnell in bestehende Software einarbeiten (bzw. wieder einarbeiten) zu können, ist gerade bei „C“ ein einheitlicher Programmierstil hilfreich. Das kann durch die Beschränkung der möglichen Sprachvielfalt unterstützt werden.

Die vorliegende Notiz enthält die Regeln für die „C“-Pro-gramm-ierung im Be-schleu-niger-Kon-troll-sys-tem. Sie wurden sehr knapp gehalten, nur das Wichtigste wurde aufgenommen. Daher sollte nicht ohne Überlegung von diesen Regeln abgewichen werden.

Die Empfehlungen gelten zunächst nur für „C“-Programme. Bei einigen Punkten sind sie aber schon mit der Zielrichtung auf „C++“ formuliert.

1.2 Wichtung

Diese Notiz ist nicht als uneingeschränkt verbindliches Regelwerk anzusehen. In begründeten Fällen kann durchaus davon abgewichen werden. Es soll aber niemals ohne Überlegung anders verfahren werden, als es hier dargelegt ist[1]. Zur besseren Beurteilung, wie gründlich über Abweichungen nachgedacht werden soll, wird eine Unterscheidung nach der Bedeutung getroffen:

Vorschrift: Ausgedrückt durch *muss* bzw. *darf nicht*.

> Vorschriften sind einzuhalten. Abweichungen davon sind nicht zulässig. Erweist sich eine Vorschrift als unpraktikabel, darf nicht kommentarlos davon abgewichen werden, sondern sie ist dann anzupassen.

Empfehlung: Ausgedrückt durch *soll* oder *darf*.

> Empfehlungen sind möglichst einzuhalten. Abweichungen davon sind nur nach ausreichender Überlegung zulässig.

Vorschlag: Ausgedrückt durch *sollte*.

> Vorschläge sollen einen Rahmen vorgeben, damit nicht über alle vorkommenden Einzelfälle nachgedacht werden muss. Da Vorschläge die weniger kritischen Punkte betreffen, sind Abweichungen möglich, wenn sie im Einzelfall günstiger erscheinen.

[1] Wenn sich eine Regel als unpraktisch erweist, muss halt die Regel angepasst werden.

2 Namensgebung

2.1 Verwendung des Unterstrichs

- Namen *dürfen nicht* mit einem „Unterstrich“ (_) beginnen und *sollen nicht* damit enden. Solche Bezeichnungen sind für systeminterne Namen reserviert (Compiler, Linker, Betriebssystem).

  Das gilt nicht mehr für Membervariablen von Klassen („C++“), da hier die Eindeutigkeit von Namen durch die Klassenbezeichnung hergestellt wird.

- In „C++“ *sollen* Membervariablen durch einen vorangestellten Unterstrich gekennzeichnet werden.

  ```
  class MyClass {
  public:
    void setClassMember(int& value) {_classMember = value;};
  private:
    int _classMember;
  }
  ```

  Das ist kein Widerspruch zur vorherigen Empfehlung: Da die Membervariablen einer Klasse zusätzlich über den Klassennamen identifiziert werden, besteht hier bei der Verwendung eines vorangestellten Unterstriches keine Gefahr der Namensgleichheit mit Systemnamen.

- Abgesehen von Bezeichnungen rein aus Großbuchstaben *sollen keine* Unterstriche innerhalb von Namen verwendet werden. Zur Abtrennung von Wortbestandteilen *sollen* die Anfänge von Wortbestandteilen mit einem Großbuchstaben beginnen, während ansonsten Kleinbuchstaben zu verwenden sind.

- Wortbestandteile aus reinen Großbuchstaben (etwa Abkürzungen wie z. B. SIS, ESR) *sollten* vermieden werden bzw. *sollen* durch Kleinbuchstaben ersetzt werden.

  ```
  UWord  myErrorMask = 0;
  UWord* myErrorMaskPtr = &myErrorMask;
  SWord  sisEsrVariant = -1;
  ```

2.2 Kennzeichnung von Sprachelementen

- Die Sprachelemente Konstanten, Typen, sowie Variablen und Prozeduren werden unterschiedlich gekennzeichnet:

  Konstanten: Konstanten *sollen* in Großbuchstaben geschrieben werden. Trennzeichen zwischen Namensteilen ist der Unterstrich (_). Elemente eines Aufzählungstyps sind ebenfalls Konstanten.

  ```
  #define MAX_LOG_DEVICE 10
  ```

```
const ULong MIL_TIMEOUT = 5;
enum Color {RED, GREEN, BLUE};
```

Typnamen, Klassennamen: Typnamen und Klassennamen *sollen* in Kleinbuchstaben geschrieben werden mit Großbuchstaben zur Abtrennung von Wortbestandteilen.

Typ- und Klassenamen *sollen* mit einem Großbuchstaben beginnen.

```
typedef ULong unsigned long;

class MyClass {
  ULong memberFunction();
   . . .
  UWord _memberVariable;
};
```

Variablennamen, Prozedurnamen: Namen von Variablen und Prozeduren *sollen* in Kleinbuchstaben geschrieben werden mit Großbuchstaben zur Abtrennung von Wortbestandteilen.

Variablennamen und Prozedurnamen *sollen* mit einem Kleinbuchstaben beginnen.

```
ULong setMyLogDev(UWord logDev){
  UWord myLogDev = logDev;
  . . .
};
```

2.2.1 Namensgebung bei Pointern

Pointer sind einerseits äußerst nützliche Sprachkonstrukte, andererseits birgt die Verwendung besondere Gefahren, da beliebige Speicherlemente überschrieben werden können, ohne dass der Compiler Möglichkeiten einer Überprüfung hat. Daher erscheint es zweckmäßig, wenn Pointer schon an ihrem Namen als solche erkenntlich sind.

- Variablen, die einen Pointer bezeichnen, *sollen* durch ein nachgestelltes großes „P“ gekennzeichnet werden.
- Wenn der Name aus der Umsetzung von Pascal-Sourcen resultiert, *darf* die bisherige Kennzeichnung beibehalten werden in den Fällen:
  - Kennzeichnung durch ein vorangestelltes kleines „p“.
  - Kennzeichnung durch „Add“ am Ende (für address).

  Andere Schreibweisen *sollen* abgeändert werden, so dass der Name mit einem großen „P“ endet.

2.3 Variablendefinitionen

Die folgende Schreibweise hat sich bei „C++“ eingebürgert, soll aber schon bei „C“ beachtet werden. Sie bietet neben anderem den Vorteil, dass die Schreibweise der Pointer der bei Pascal gewohnten entspricht.

- Jede Variable *muss* in einer eigenen Definition angelegt werden. Also

  ```
  UWord ind;
  UWord cnt;
  ```

  anstelle von `UWord ind, cnt;`.

- Bei der Definition/Deklaration von Pointern *soll* der '`*`' direkt an den Typnamen angehängt werden. Entsprechend *soll* das '`&`' bei Referenzen direkt an den Typnamen angehängt werden.

  ```
  ULong* statusPtr = NULL;
  CHAR* str;
  void getValue(UWord& data) { . . .}
  ```

  Hierfür ist zwingend erforderlich, dass pro Deklaration nur eine Variable aufgeführt ist.

2.4 Konstantendeklarationen

Konstanten können per `#define`-Anweisung oder als konstante Variablen angelegt werden (`const int`). Günstiger ist die Deklaration als konstante Variable, da dann eine Typinformation gegeben ist und der Name auch einem Debugger bekannt ist.

- Konstanten *sollen* als konstante Variable angelegt werden.

  ```
  const ULong BUFFER_SIZE = 0x100; // VMS only
  ```

Zur Schreibweise (in Großbuchstaben) siehe Abschn. 2.2.

2.5 Verwendung von Aufzählungstypen

- Kann eine Variable nur wenige ganzzahlige Werte annehmen, *sollen* dafür Aufzählungstypen und keine einzeln definierten Konstanten verwendet werden.
- Aufzählungstypen *dürfen nicht* zur Variablendefinition verwendet werden, wenn es um den Austausch zwischen verschiedenen Komponenten geht (siehe Abschn. 3.3).
- Aufzählungstypen *sollen* als erstes Element (mit dem Wert 0) ein nicht verwendetes Element haben (`ENUM_UNDEF`).

  ```
  enum MyChoice {MY_CHOICE_UNDEF, MY_CHOICE_NO, MY_CHOICE_YES};

  UWord choice = MY_CHOICE_YES;
  ```

2.6 Strukturen

- Strukturen sollen vor der Verwendung in einer Definition über eine `typedef`-Anweisung deklariert werden:

```
typedef struct {
  Float32 realPart;
  Float32 imagPart;
} Complex;

Complex complArray [42];
Complex cmpl;
```

  Wenn die Struktur an mehreren Stellen verwendet wird, erfolgt die Definition per `typedef` zu Beginn des Programms oder, wenn sie von mehreren Moduln benötigt wird, in einer Header-Datei.

3 Definierte Speichergröße der Variablen

3.1 Austausch zwischen Komponenten: Einheitliches Speicherlayout

Wenn Informationen zwischen verschiedenen Komponenten ausgetauscht werden soll, (zwischen SW-Modulen oder wenn HW per SW angesprochen wird), muss natürlich von beiden Seiten dasselbe Datenformat verwendet werden. Das wird besonders deutlich, wenn komplexe Strukturen ausgetauscht werden.

Leider ist die Größe der elementaren Speichertypen nicht eindeutig definiert. Es gelten lediglich Größenbeziehungen (`long` ist nicht kleiner als `int` und `int` nicht kleiner als `short`, `double` ist nicht kleiner als `float`. Ob dageben der Typ `int` durch eine 16-, 32- oder 64-Bit Zahl repräsentiert wird, hängt vom verwendeten Compiler und der Prozessorhardware ab.

Ähnliches gilt für Aufzählungstypen: Auch hier ist nicht festgelegt, wie dieser Typ im Speicher repräsentiert wird (8, 16, 32 Bit).

Betroffen sind zumindest drei Schnittstellen:

DPR: Das ist derzeit zwar noch kein Problem, da sowohl für SE als auch GμP Compiler und Prozessorfamile dieselben sind.

Werden aber einmal unterschiedliche Compiler verwendet (etwa, weil unterschiedliche Prozessoren zum Einsatz kommen), kann dieses Problem schlagartig auftreten.

Netzwerk-Packete: Betrifft in der Geräte-SW die Sende- und Empfangsstrukturen der USRs. Schon jetzt müssen die Datenstrukturen der VME-Ebene auf der andersartigen VMS-Ebene gleichartig interpretiert werden.

Zukünftig soll erreicht werden, dass von (weitgehend) beliebigen Plattformen mit den VME-Rechnern kommuniziert werden kann. Auf allen diesen Rechnern und Betriebssystemen (z. B. PCs unter Windows und Linux, Compaq-AXP unter Linux

und VMS) müssen dann die Datenstrukturen gleich interpretiert werden.

Hardware-Register: Bei Zugriffen auf Hardware ist die Speichergröße durch die Breite der Hardware-Register vorgegeben. Bei Zugriffen über den MIL-BUS ist das eigentlich immer eine Größe von 16 Bit.

Es sollte erreicht werden, die Software auch dann unverändert weiternutzen zu können, wenn der Datentyp `short` einmal durch 32 Bit und nicht mehr 16 Bit abgebildet werden sollte.

Es ist daher dafür zu sorgen, dass das Aussehen der Austauschstrukturen unabhängig vom aktuell verwendeten Compiler und der aktuellen Hardware (Prozessor) beschrieben wird.

3.2 Selbstdefinierte Typen

- Bei allen Variablen, die zum Austausch zwischen verschiedenen Komponenten benutzt werden (DPR, Netzwerkpackete, Hardware-Zugriffe) *dürfen keine* elementaren Datentypen verwendet werden.
- Es *müssen* stattdessen die in zentralen Includes deklarierten Typen verwendet werden:

```
#include "lib68:global_types.h"
UWord   vAcc;
UWord   lDev;
```

- Bei rein lokal verwendeten Variablen wie Schleifenzählern können elementare Datentypen verwendet werden.

```
for (int i = 0; i < NAME_LENGTH; i++) {
  dest[i] = source[i];
}
```

3.3 Aufzählunsgtypen

- Aufzählungstypen *dürfen nicht* zur Variablendefinition verwendet werden, wenn es um den Austausch zwischen verschiedenen Komponenten geht, da die Größe, mit der solche Variablen angelegt werden, vom Compiler abhängt.

  Zum Austausch zwischen verschiedenen Komponenten sind nur Variablen definierter Größe zu verwenden (`SWord`, `ULong`, ...).

  Wegen der Typkompatibilität können solchen Variablen dann aber Werte aus Aufzählungstypen zugewiesen werden.

```
enum MyChoice {MY_CHOICE_UNDEF, MY_CHOICE_NO, MY_CHOICE_YES};
UWord choice = MY_CHOICE_YES;
```

4 Formatierung

4.1 Einzüge

- Einzüge sollen jeweils zwei Zeichen betragen. Es sollen Leezeichen an Stelle von TABs verwendet werden

4.2 Block-Klammerung

- Die einleitende Klammer *soll* in der gleichen Zeile stehen wie die den Block einleitende Anweisung (Laufanweisug, Abfrage). Die den Block beendende Klammer *muss* in der gleichen Einrückungsebene stehen wie der den Block einleitende Befehl.

```
for (i = 0; i < 8; i++) {
  if ((i & 1) == 0) {
    j += i;
  }
}

if (a == 0) {
  b = 0;
}
else {
  b = a;
}
```

4.3 Leerzeichen

- Die meisten binären Operatoren *sollen* durch Leerzeichen abgetrennt werden. Dadurch wird die Lesbarkeit erhöht.

```
a = b + c;
for (int i = 0; i < 1; i++) {
  j = 2 * i;
}
```

- Die wesentliche Ausnahme sind Operatoren zur Memberauswahl, hier *sollen* Strukturname und Strukturelement durch die Operatoren verbunden werden:

```
structure.element
structPtr->element
```

5 Formulierungen

5.1 Variablendefinitionen

- Alle Variablen *sollen* bei ihrer Definition initialisiert werden:

```
UWord  myVariable = 0;
UWord* myVariablePtr = &myErrorMask;
```

- Zentrale Variablen oder mehrfach verwendete Variablen *sollten* zu Beginn der Routine definiert werden.

```
void currentSEqm (. . .) {
  DprType* const dpr = (DprType*) &dprbase_;
  ErrorType cStatus = -1;
  UWord wrtCount = 0;
  . . .
  readAndUpdateStatus(logDev, ifbAdr, cStatus);
  . . .
  writeMil(. . ., cStatus);
  . . .
```

- Variable, die nur an einer Stelle verwendet werden, *sollten* unmittelbar vor ihrer Verwendung definiert werden.

```
. . .
int i = 0;  // needed for loop only
for (i = 0; i < NOMEN_LENGTH; i++) {
  dest [i] = source[i];
}
. . .
```

5.2 Äquivalenz zwischen Pointer und Array

In der Sprache C werden Arrays und Pointer weitgehend gleichwertig behandelt. Insbesondere können einzelne Array-Elemente direkt über Pointer adressiert werden. Die beiden folgenden Zugriffe sind identisch:

```
SLong arr[8];
arr[3]   = 17;  // array element 3
*arr + 3 = 17;  // 3 times size of pointer type behind start of arr
```

Obwohl beide Schreibweisen identische Zugriffe beschreiben, wird zumindest für Personen mit Pascal-Hintergrund bei der ersten eher suggeriert, dass Operationen auf Feldern durchzuführen sind. Daher gilt:

- Wenn es sich um Zugriffe auf Array-Strukturen handelt, *soll* die Schreibweise als Felder gewählt werden (mit eckigen Klammern).
- Wenn es sich um Zugriffe auf einzelne Strukturen handelt (so wie Pointer in Pascal verwendet werden), *soll* die Schreibweise als Pointer verwendet werden (mit dem Stern- bzw. Pfeil-Operator).
- Bei der Übergabe von Feldern als Prozedur-Parameter *sollte* ebenfalls die Schreibweise als Felder gewählt werden. Eine Prozedur mit einem Feld als Parameter ist also nicht zu deklarieren als `void proc(SLong* arr)`, sondern *sollte* deklariert werden als

  ```
  void proc(SLong arr[])
  ```

  Natürlich wird in beiden Fällen nur ein Pointer auf den Beginn des Arrays übergeben, die spezielle Schreibweise legt aber nahe, dass es sich um ein Feld handelt.

Wenn Arrays konsequent als solche geschrieben werden (mit eckigen Klammern), sollte sich die undurchsichtige Pointer-Arithmetik weitgehend vermeiden lassen.

5.3 Parameterdeklaration in Prozeduren

Das originale „C“ bietet nur den Übergabemechanismus „call by value“. Wenn ein Übergabeparameter durch die Prozedur verändert werden soll, muss ein Pointer auf die Variable übergeben werden. Innerhalb der Prozedur müssen dann alle Zugriffe über diesen Pointer erfolgen. Mit der Verwendung eines „C++“-Compilers (also insbesondere in „C++“-Code) steht aber die Möglichkeit der Variablenübergabe per Referenz zur Verfügung. Diese Möglichkeit ist der Übergabe eines Pointers vorzuziehen:

- Werden Varialblen durch eine Prozedur verändert, *sollen* diese Variablen der Prozedur nicht über einen Pointer übergeben werden, sondern als Referenz:

  ```
  void invertIntParameter(int& para) {
    para = -1 * para;
  }
  ```

- Zur Verbesserung der Laufzeit *sollten* Prozedurparameter, die größere Strukturen darstellen, auch dann per Referenz übergeben werden, wenn sie durch die Prozedur nicht verändert werden. Zur Sicherheit *sollen* sie dann als konstant deklariert werden:

  ```
  typedef struct {
    int a;
    int b[37];
    float c;
  } BigParameter;

  int evaluateBigParameter(const BigParameter& para) {
    . . .
  }
  ```

6 Sonstiges

6.1 Includes

- In C++ *sollen* die C++-spezifischen Header-Dateien inkludiert werden. Sie enthalten die selben Definitionen wie die C-spezifischen Header-Dateien, befinden sich aber innerhalb des Namespaces `std`.

  Beispiel: Statt `#include <time.h>` soll in C++ `#include <ctime>` benutzt werden.

C Include	C++ Include	C Include	C++ Include	C Include	C++ Include
assert.h	cassert	locale.h	clocale	stdio.h	cstdio
ctype.h	cctype	math.h	cmath	stdlib.h	cstdlib
errno.h	cerrno	setjmp.h	csetjmp	string.h	cstring
float.h	cfloat	signal.h	csignal	time.h	ctime
iso646.h	ciso646	stdarg.h	cstdarg	wchar.h	cwchar
limits.h	climits	stddef.h	cstddef	wctype.h	cwctype

  Siehe auch das Skript `c2cpp-includes <file> {<file>}`, das alle C-Inkludes in C++-Inkludes in den angegebenen Dateien umsetzt .

6.2 Typwandlungen

- In C++ *sollen* zur Typwandlung nur noch die *Typwandlungsoperatoren* benutzt werden. Es sind dies `static_cast`, `dynamic_cast`, `const_cast` und `reinterpret_cast`.

```
int i;
char c;

i = (int)c;                // alt, nicht mehr benutzen
i = static_cast<int>(c);   // besser so
```

6.3 Trickprogrammierung vermeiden

- Zuweisungen innerhalb von Anweisungen *sollen* nur dann verwendet werden, wenn sehr klar ist, was dort passiert. Beispiel:

```
while ((c = getchar()) != EOF) {
  /* do something */
}
```

- Sehr komprimierte Anweisungen *sollen* nicht verwendet werden. Beispiel für zu kompakten Code:

```
a += (a *= b) % c;
i+= (i >= j >= --k) ? j++ : k;
```

- Geniale C-Konstrukte *sollten* vermieden werden, wie

  ```
  while (dest++ = src++);
  ```

6.4 Membervariablen in C++

- In einer Klassendeklaration *sollen* Membervariablen durch einen vorangestellten Unterstrich gekennzeichnet werden:

  ```
  class MyClass {
  public:
    MyClass();
    ~MyClass();
  private:
    int    _cnt;
    double _value;
  }
  ```

  Dadurch ist in einer Methode leicht zu erkennen, welche Variablen zu der Klasse gehören und welche lokal deklariert sind oder als Parameter übergeben werden.

Index

— **V** —

— **W** —