Einführung in C & C11 (ISO/IEC 9899:2011) Spezifikation

Seminar: Effiziente Programmierung in C

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Daniel Martens
E-Mail-Adresse:	danielmartens01@googlemail.com
Matrikelnummer:	6511159
Studiengang:	Informatik B.Sc.

Hamburg, den 26.03.2014

Einleitung

Dieses Dokument beinhaltet die Ausarbeitung zum meinem Vortrag zum Thema C11. Den Vortrag habe ich in dem Seminar “Effiziente Programmierung in C” im WiSe 2013/2014 gehalten. Da ich der erste Vortragende in der Vortragsreihe war, enthält diese Ausarbeitung, wie mein Vortrag auch, eine allgemeine Einführung in C.

Einführung in C

Dieses Kapitel klärt die Entstehung der Programmiersprache C. Es geht auf die Evolution der Sprache ein und nennt die Vorzüge. Es werden ausserdem Anwendungsgebiete der Sprache genannt.

Dem Kapitel vorausgestellt ist ein Hello-World Beispiel.

Listing 1: Hello-World in C.

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     printf("Hello world!\n");
6
7     return 0;
8 }
```

Entstehung und Evolution

Die Programmiersprache C wurde 1969 bis 1973 in den AT&T Bell Labs von Dennis Ritchie entwickelt. Bei C handelt es sich um eine imperative Programmiersprache. Sie wurde von anderen Programmiersprachen, wie B, BCPL und ALGOL 68 beeinflusst.

1973 ist die Sprache so weit ausgereift, dass der Unix-Kernel der PDP-11 in C neu geschrieben werden kann. Der erste C-Compiler der dazu notwendig war wurde ebenfalls von Dennis Ritchie entwickelt. Zu dieser Zeit gibt es allerdings noch keine Spezifikation zu der Programmiersprache.

Die Spezifikation wird 1978 von Brian W. Kernighan und Dennis Ritchie in einem Buch veröffentlicht und trägt den Titel “The C Programming Language”.

C ist damit bis heute eine der wenigen Sprache die eine Spezifikation besitzen. Zu den bis heute entstandenen Spezifikationen der C Programmiersprache zählen C89/90, C95, C99 und C11.

Vorteile

Ein besonderer Vorteil von Programmen in C ist ihre Portierbarkeit. C-Compiler gibt es für sehr viele Prozessoren, ausserdem sichert die C-Spezifikation zu, dass Quelltext auf allen Maschinen funktionieren muss, also unabängig von der zugrunde liegenden Hardware.

Es gibt zwar dennoch Abwandlungen von C, eine Spezifikation ermöglicht es aber erst zuzusichern, dass Quelltext der entlang der Spezifikation programmiert ist auf allen Maschinen laufen kann.

Ein weiterer Vorteil ist die Geschwindigkeit von C. C ist sehr performant und läuft beispielsweise, im Gegenteil zu Java, nicht in einer virtuellen Maschine, kann also direkt vom Prozessor ausgewertet werden.

Neben der Geschwindigkeit ist auch die Grösse von C ein Vorteil, diese ist minimal.

C kann verwendet werden um sehr systemnah zu Entwickeln. Es ermöglicht Low-Level-Access, man kann also die Hardware direkt ansprechen. Das ist auch einer der Gründe warum C schon lange in der Entwicklung von Betriebssystemen eingesetzt wird und auch bei Mikrocontrollern.

Anwendungsgebiete

C wird unter anderem dank seiner Portierbarkeit in vielen Bereichen eingesetzt. Wie zuvor schon genannt, wird es in der systemnahen Entwicklung eingesetzt, wie beispielsweise in der Entwicklung von Betriebssystemen.

Ein weiteres beliebtes Anwendungsgebiet von C sind Mikrocontroller. Für diese sind oft keine anderen Compiler verfügbar, ausserdem läuft der C-Code direkt auf dem Prozessor und nicht in einer virutellen Maschine. Auch die Grösse des C-Programms spielt eine wichtige Rolle hierbei. Neben Mikrocontrollern sind weitere Anwendungsgebiete auch eingebettete Prozessoren oder DSP-Prozessoren.

C wird ebenfalls als Interpreter anderer Programmiersprachen eingesetzt. So können diese von der hohen Portierbarkeit von C Gebrauch machen. Bekannte Beispiele hierfür sind u.a. Java, die Java Virtual Machine ist in C geschrieben, auch aus Perfomanzgründen.

Auch dient C als Zwischencode für einige Programmiersprachen, ein bekanntes Beispiel hierfür ist Vala.

Zusammenfassung

Zusammenfassend ist zu sagen, dass C eine der am weit verbreitetsten Programmiersprachen ist. C besitzt im Gegenteil zu den meisten Programmiersprachen eine Spezifikation. Die hohe Portierbarkeit und geringe Grösse sorgt für eine Vielzahl von Anwendungsgebieten.

1 C11 Allgemein

Dieses Kapitel, sowie das Folgende, beinhalten das Hauptthema des Vortrags, den C11 Standard. Dieses Kapitel diskutiert zuerst warum Standardisierung sinnvoll ist. Die existierenden Standards werden kurz wiederholt, im Anschluss wird auf die Regeln für Änderungen eingegangen.

1.1 Warum Standardisierung?

Die Programmiersprache C hat sich seit Beginn ihrer Entstehung sehr schnell weiterentwickelt. Die anzutreffenden Varianten von C entsprachen nicht mehr den von Kerninghan und Ritchie beschriebenen Vorgaben.

Es gab viele Erweiterungen von C, allerdings fehlte die Vereinheitlichung. Ziel der Standardisierung ist eine Normierung der Sprache.

Nur durch Standardisierung ist es möglich Quelltext zu schaffen der sehr portierbar ist, d.h. unabhängig von der zugrundeliegenden Hardware auf allen Umgebungen funktioniert.

1.2 Zeitliche Abfolge der C Standards

C wurde mehrfach standardisiert. Bereits in dieser Arbeit genannte Standards sind C89/90, C95, C99 und C11.

1983, 10 Jahre nach der Fertigstellung der Programmiersprache C nahm sich erstmals eine Komitee der Ausarbeitung eines ersten Standards an. Das Komitee trug den Namen X3J11 und gehörte zum American National Standards Institute (ANSI).

Das Komitee stellte bis 1989 den ersten Standard zusammen (ANSI X3. 159-1989 Programming Language C). Dieser Standard wird 1990 mit kleinen Änderungen von der International Standards Organization (ISO) übernommen.

1995 (C95) wird der C90 Standard von der International Standards Organization ergänzt.

1999 wird der ISO/IEC 9899 Standard (C99) verabschiedet. In diesen Standard fliessen auf C++ bekannte Erweiterungen in C zurück.

Seitdem arbeitete das Komitee WG14 an einem Standard namens C1X, der heute als C11 bekannt ist.

1.3 Regeln für Änderungen

Trotz der häufigen Standardisierung von C gibt es genaue Regelungen für Änderungen, um die Anwender der Sprache nicht zu verwirren und für grösstmögliche Kompatibilität zu sorgen.

Aus diesem Grund wird eine vollständige Kompatibilität zum vorherigen Standard versprochen. Bestehender Code hat bei Standardisierung der Programmiersprache C einen sehr hohen Stellenwert, deswegen sollen Modifikationen den Geist der Sprache nicht verändern.

Ein nennenswertes Beispiel hierfür ist, dass der Programmierer in C schon immer zusichern musste, dass der zur Verfügung gestellte Speicher das Ergebnis einer Operation aufnehmen kann. Dieses Prinzip des Vertrauens lässt sich auch anderes regeln, wird aber, da der Geist der Sprache erhalten bleiben soll, nicht verändert. Stattdessen werden optionale Spracherweiterungen, z.B. Bounds-Checking zur Verfügung gestellt. Wir gehen darauf später in den Features ein.

2 C11 Features

Dieses Kapitel ist das umfangreichste der Arbeit. Es geht auf die neuen Features der C11 Spezifikation ein. Es wird gezeigt, warum diese Änderungen eingeführt wurden. Einige Features werden mit Beispielen demonstriert.

2.1 Thread Unterstützung

Multithreading ist ein wichtiges Thema in der heutigen Zeit. Mit steigender Anzahl an Prozessorkernen ist es erforderlich auf die Vorteile des Multihreading zurückzugreifen, um effiziente Berechnungen ermöglichen zu können.

<threads.h> ist Bestandteil von C seit dem C11 Standard. In vielen Anwendungen konnte schon vor diesem Standard nicht auf Multihreading verzichtet werden. Das führt zu unterschiedlichen Implementierungen auf unterschiedlichen Plattformen. Im UNIX Umfeld wurde auf POSIX Threads zurückgegriffen während bei Windows Windows Threads verwendet wurden.

Konsequenz hieraus war, dass der Quelltext nicht mehr portierbar war. Multithreading-Code der unter UNIX geschrieben wurde, konnte nicht auf Windows eingesetzt werden. Die C Spezifikation soll eine abstrakte Maschine referenzieren, die eine Generalisierung der benutzten Maschine darstellt, d.h. die Spezifikation bezieht sich weder auf einen speziellen Compiler, noch auf ein Betriebssystem, noch auf eine CPU. Ziel des C11 Standards war es hier eine Vereinheitlichung herbeizuführen um die Portierbarkeit gewährleisten zu können.

Bei diesem Feature hat man sich entschieden, sich an die POSIX Threads anzulehnen, da diese bereits gut dokumentiert waren. Eine Neuentwicklung war nicht sinnvoll, da man nach den Regeln der Änderungen den Anwender nicht verwirren wollte.

2.1.1 Methoden

An dieser Stelle möchte ich drei Methoden exemplarisch vorstellen, die für das Multihreading eingeführt wurden.

Listing 2.1: Multithreading. Thread erzeugen.

```
1 int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

Mit thrd_create lässt sich ein neuer Thread erzeugen. Der erzeugte Thread führt daraufin func(arg) aus. Wird der Thread erfolgreich erzeugt, ist thr der Identifier des neu erzeugten Threads.

Listing 2.2: Multithreading. Thread beenden.

```
1 _Noreturn void thrd_exit(int res);
```

Mit thrd_exit lässt sich ein Thread beenden. Das Resultat des Threads wird dabei an res übergeben. Das Programm insgesamt terminiert, nach Beenden des letzten Threads, normal.

Listing 2.3: Multithreading. Threads zusammenführen.

```
1 int thrd_join(thrd_t thr, int *res);
```

Threads lassen sich ebenfalls mit thrd_join zusammenführen. Die Methode führt den aktuellen Thread mit dem in thr referenzierten Thread durch Blockieren zusammen, bis der Thread terminiert ist. res ist entweder ein Nullpointer oder enthält das Resultat des zusammengeführten Threads.

2.2 Atomare Operationen

Atomare Operationen stellen ein weiteres Feature des C11 Standards dar. Diese waren zwar schon vorher möglich, durch Rückgriff auf externe Bibliothek, wie beispielsweise GLib, aber nicht Teil des Standards. Ihre Verwendung war also an Abhängigkeiten geknüpft.

Die Problemstellung lautet, dass mehrere Threads dieselben Variablen nutzen sollen. Die Operationen sollen serialisiert werden. Schauen wir uns das gegebene Problem, dem entgegengewirkt wurde, vor der C11 Spezifikation an.

Gegeben sind zwei Integer-Variablen a und b.

Thread 1	Thread 2
a = 1;	printf(ä = %d; a);
b = 2;	printf(b̈ = %d; b);

Tabelle 2.1: Atomare Operationen vor C11 Standard

Ausgehend von diesem Beispiel ist das Verhalten vor C11 unspezifiziert, da es keine Spezifikation für Threads gibt.

Es gibt zwar die Möglichkeit mit Mutexen zu arbeiten, beispielsweise

Listing 2.4: Atomare Operationen. Mutex.

```
1 mtx_lock(&mutex);
2 a=1;
3 mtx_unlock(&mutex);
```

allerdings ist diese Form für einfache Operationen zu komplex.

Es gibt zwar Prozessoren die solche Operationen als einen Schritt durchführen können, im Allgemeinen sind solche Operationen aber sehr teuer, da der Speicherzugriff für andere Prozesse kurzzeitig blockiert werden muss.

C11 abstrahiert diese Prozessorbefehle und stellt <stdatomic.h> zur Verfügung. In <stdatomic.h> werden Typen, Makros und Funktionen für atomare Operationen bereitgestellt. Atomare Operationen werden auch häufig im Linux-Kernel verwendet, da sie performanter als Lockingmechanismen sind.

2.2.1 Methoden

An dieser Stelle möchte ich wieder drei Methoden für atomare Operationen vorstellen, die wir benötigen, um unser Beispiel unter C11 Spezifikationen fortführen zu können.

Listing 2.5: Atomares Objekt initialisieren.

```
1 void atomic_init(volatile A *obj, C value);
```

atomic_init initialisiert ein atomares Objekt des Zeigers obj mit dem Wert aus value.

Beispielaufruf.

Listing 2.6: Atomares Objekt initialisieren. Beispiel.

```
1 atomic_int a;
2 atomic_init(&a, 1);
```

Listing 2.7: Wert für atomares Objekt setzen.

```
1 void atomic_store(volatile A *object, C desired);
2
3 void atomic_store_explicit(volatile A *object, C desired,
    ↪ memory_order order);
```

atomic_store ersetzt den Wert von object mit dem Wert aus desired. order gibt dabei das Verhalten beim Ersetzen an.

Wir gehen gleich auf das order Parameter ein.

Listing 2.8: Wert eines atomaren Objekt laden.

```
1 C atomic_load(volatile A *object);
2
3 C atomic_load_explicit(volatile A *object, memory_order
    ↪ order);
```

atomic_load lädt den Wert eines Objektes und gibt ihn zurück. order gibt dabei an wie der Wert geladen werden soll.

2.2.2 Beispiel unter C11 Standard

Wie im vorangegangenen Beispiel verwenden wir wieder zwei Integer-Variable, diesmal atomarer Form.

Listing 2.9: Atomare Integer-Variablen.

```
1 atomic_int a; atomic_int b;
```

Wir übertragen nun das vorangegangene Beispiel, unter Verwendung der eben erklärten Methoden, in die C11 Spezifikation.

Thread 1	Thread 2
atomic_store(&a, 1);	atomic_load(&b);
atomic_store(&b, 2);	atomic_load(&a);

Tabelle 2.2: Atomare Operationen vor C11 Standard

Seit der C11 Spezifikation ist dieses Verhalten nun erstmals definiert.

Allerdings gibt es immernoch drei mögliche Varianten wie bei unserem Beispiel verfahren werden kann. Thread 2 lädt

- b = 0 und a = 0, wenn dieser vor Thread1 ausgeführt wird
- b = 2 und a = 1, wenn dieser nach Thread1 ausgeführt wird
- b = 0 und a = 1, wenn dieser ausgeführt wird nachdem Thread1 a gespeichert hat und bevor b gespeichert wurde.

Der folgende Fall kann nicht eintreten

- b = 2 und a = 0, weil die Standardeinstellungen für atomares Laden und Speichern sequentielle Konsistenz vorschreibt.

Das bedeutet gleichzeitig, dass die Operationen in der Reihenfolge geschehen müssen, wie sie in den Threads beschrieben sind.

2.2.3 Beispiel mit Verbesserungen

Der Nachteil an dem oben genannten Beispiel ist, dass dieses auf einer CPU sehr teuer ist.

Es gibt allerdings einen Weg dies performanter zu gestalten. Nehmen wir an, dass der Algorithmus keine Ordnung benötigt, also beispielsweise den Zustand b = 2 und a = 0 akzeptiert. Dann kann für das Laden und Speichern eine bestimmte Ordnung angegeben werden.

Durch das Setzen des order Flags ist das Verhalten auf modernen CPUs wesentlich schneller.

Thread 1	Thread 2
atomic_store(&a, 1, memory_order_relaxed); atomic_store(&b, 2, memory_order_relaxed);	atomic_load(&b, memory_order_relaxed); atomic_load(&a, memory_order_relaxed);

Tabelle 2.3: Atomare Operationen vor C11 Standard, verbessert.

Es gibt noch weitere order Flags die ich hier kurz aufführen möchte. Dazu zählen memory_order_consume, memory_order_acquire, memory_order_release und memory_order_acq_rel.

Flag	Funktion
memory_order_acquire	Garantiert, dass nachfolgende Aufrufe nicht durchgeführt werden bevor der aktuelle Aufruf und Vorangegangene erledigt wurden.
memory_order_consume	Schwächere Form von memory_order_acquire.
memory_order_release	Garantiert, dass vorangehende Speicherungen nicht nach dem aktuellen Speichern erfolgen oder später.
memory_order_acq_rel	Vereinigt memory_order_acquire und memory_order_release.
memory_order_relaxed	Alle Ausführungsreihenfolgen sind erlaubt.

Tabelle 2.4: Atomare Operationen, Order-Flags.

2.3 Bounds-Checking

Bounds-Checking ist im Annex K des ISO/IEC 9899:2011 Standard zu finden. Es stellt eine sicherheitsrelevante Erweiterung dar. Durch die Erweiterung soll sichergestellt werden, dass genügend Speicher für das Ergebnis einer Operation zur Verfügung steht. Die Zusicherung lag vorher beim Entwickler und hat zu Buffer-Overflows geführt.

Seit C11 gibt es unterstützende Methoden mit dem Suffix _s, um vor Sicherheitslücken zu schützen.

Im Folgenden wollen wir wieder drei der Methoden exemplarisch Nennen und Erläutern.

2.3.1 Methoden

Listing 2.10: Bounds-Checking. strcat.

```
1 errno_t strcat_s(char * restrict s1, rsize_t s1max, const
    ↪ char * restrict s2);
```

strcat_s erwartet einen zusätzlichen Parameter für die Puffergrösse und stellt sicher, dass nicht mehr als s1max Bytes zu s1 kopiert werden.

Listing 2.11: Bounds-Checking. strcopy.

```
1 errno_t strcopy_s(char * restrict s1, rsize_t s1max, const
    ↪ char * restrict s2);
```

strcopy_s stellt sicher, dass s1max grösser ist als die Länge von s2.

Listing 2.12: Bounds-Checking. gets.

```
1 char *gets_s(char * restrict buffer, size_t nch);
```

gets_s liest höchstens nch Zeichen von der Standardeingabe. gets war zuvor eine der am häufigsten genutzten Sicherheitslücken, deswegen wurde es in C99 bereits deprecated und in C11 schliesslich vollständig entfernt.

Die Methoden sind an Visual C angelehnt. Dort gab es bereits vorher Methoden mit dem Suffix _s. Die Implementierung in C11 ist ähnlich aber nicht identisch.

2.4 Static Assertions

Bei Static Assertions handelt es sich um Ausdrücke, die zur Übersetzungszeit ausgewertet werden. Wird der Ausdruck zu 0 ausgewertet, wird ein String/Fehler ausgegeben. Static Assertions sind besonders hilfreich bei Debuggen bzw. bei der Fehleranalyse. Sie sind ein schon lange aus Java bekanntes Sprachkonstrukt.

Zur Zeit der C89 Spezifikation wurden Tests vom Präprozessor ausgeführt. Die Ausgabe der Fehler geschah durch die Präprozessoranweisung #error. Allerdings sind Tests durch den Präprozessor sehr limitiert.

Der Befehl sizeof wird beispielsweise nicht korrekt ausgewertet, was daran liegt, dass diese Methode, wie andere auch, erst konvertiert wird, wenn der Präprozessor ausgeführt wurde.

2.4.1 Tests mit Präprozessor in C89

Listing 2.13: Tests mit Präprozessor in C89

```
1 #if __STDC__ != 1
2 #error "Fehlerbeschreibung"
3 #endif
4
5 #if sizeof(long) < 8
6 #error "Fehlerbeschreibung"
7 #endif
```

Was benötigt wird ist eine Anweisung, die diese Funktion zur Übersetzungszeit übernimmt, nachdem der Präprozessor ausgeführt wurde.

2.4.2 _Static_assert

Seit C11 stellt _Static_assert diese Funktion bereit.

_Static_assert besteht aus einem Ausdruck, der zur Übersetzungszeit ausgewertet wird sowie einem String, der ausgegeben wird wenn die Anweisung zu 0 (Fehlerfall) ausgewertet wird.

Listing 2.14: _Static_assert. Beispiel.

```
1 _Static_assert(constant-expression, string-literal);
2
3 _Static_assert(sizeof(size_t) >= 8, "Fehlerbeschreibung");
```

2.5 Weitere Neuerungen

C11 stellt noch wesentlich mehr Neuerungen zur Verfügung, auf die hier aus Gründen des Umfangs nicht eingegangen wird. Dazu zählen

- Exklusiver Dateizugriff
- Generic-Selection
- Unicode-Unterstützung
- Speicherzugriffe und Speicherausrichtung
- ...

2.6 Compiler-Unterstützung

Zum Schluss möchte ich noch einen Überblick über die Compilerunterstützung des C11 Standards geben, da nicht alle Compiler (zur Zeit des Vortrags) die Spezifikation vollständig unterstützen. Gleiches gilt ebenfalls für C99, noch heute.

	C99	C11
LLVM/Clang	Nahezu alle Features (keine C99 Floating Point Pragmas)	Teilweise
Pelles C	Vollständige Unterstützung	Vollständige Unterstützung
IBM XL	Vollständige Unterstützung	Vollständige Unterstützung
GCC	Vollständige Unterstützung	Teilweise

Tabelle 2.5: C11 Compiler Unterstützung.