MASTERARBEIT

zur Erlangung des akademischen Grades
„Master of Science in Engineering“
im Studiengang Embedded Systems

Der Objekt Orientierte Ansatz in der Entwicklung von Eingebetteten Systemen

Ausgeführt von: Ney Fränz, BSc

Personenkennzeichen: 1610297013

BegutachterIn: FH-Prof. Dipl.-Ing. Dr. Martin Horauer

Wien, den 15. Juli 2018

FH University of Applied Sciences
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. Urheberrechtsgesetz idgF sowie Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtoolasas entspricht.“

Wien, 15. Juli 2018 Unterschrift

Kurzfassung

In der Entwicklung von eingebetteten Systemen hat sich in den letzten Jahren einiges getan. Moderne Programmiersprachen wie C/C++ haben sich in der embedded Entwicklung etabliert und aufwendiges programmieren in Assembler sollte nur noch in wenigen Fällen von Nöten sein. Diese Arbeit beschäftigt sich hauptsächlich mit der Frage, ob der Einsatz einer objektorientierten Programmiersprache auf Plattformen mit nur wenigen kBytes an Flash Speicher sinnvoll ist und welchen Mehrwert diese für die Embedded Entwicklung haben könnte.

Hierbei sollen vor allem die gängigsten Konzepte (Klassen, Templates, etc.) der objektorientierten Sprache analysiert werden, um Solide Richtwerte über Performance und Speicherverbrauch geben zu können. Dazu soll der kompilierte Code analysiert und diverse Benchmark Tests durchgeführt werden. Zusätzlich wird der Vergleich mit einer klassischen funktionalen Programmiersprache dargestellt.

Als Referenz Programmiersprache wird C/C++ in Verbindung mit der ARM Cortex-M Architektur verwendet, da diese Kombination sehr interessant für stromsparende und kleinere IoT Projekte ist und sich wahrscheinlich in Zukunft durchsetzen wird. Am Anfang wird auch eine State-Of-the-art Analyse über die momentan verfügbaren ARM C++ Compiler durchgeführt um auflisten zu können welche Versionen und Erweiterungen von C++ unterstützt werden.

Schlagworte: Objektorientiertes Programmieren, C/C++, Embedded Software, ARM Cortex-M

Abstract

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Keywords: Keyword1, Keyword2, Keyword3, Keyword4

Danksagung

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Inhaltsverzeichnis

1 State-of-Art Analyse **1**

1.1 Mikrocontroller Hardware Analyse 1

1.1.1 Stromverbrauch und Effizienz 1

1.1.2 MCU Peripherie 1

1.1.3 Sicherheits- Features 1

1.2 Work-Flow und Tools in der embedded Entwicklung 1

1.2.1 Entwicklungs- Umgebungen 1

1.2.2 Test und Qualitäts- Management 1

1.2.3 Code Generatoren 1

1.2.4 Low-Level Hardware Bibliotheken 1

1.3 Programmiersprachen für die embedded Entwicklung 1

1.3.1 Kompilierte Programmiersprachen 1

1.3.2 Interpretierte Programmiersprachen 1

1.4 Verfügbare C/C++ Compiler für die ARM Architektur 1

1.4.1 GNU Arm Embedded Toolchain 1

1.4.2 IAR Embedded Workbench 1

1.4.3 ARM Compiler 1

2 Embedded C++ **1**

2.0.1 Überblick 2

2.0.2 C++ Konzepte 2

2.0.3 Eingabe- und Ausgabestreams 2

2.0.4 'auto'-Typ 3

2.0.5 Funktionen Überladen 4

2.0.6 Funktions- Templates 6

2.0.7 Arrays 8

2.0.8 Pointer vs Referenzen 10

2.0.9 Namespaces 10

2.0.10 Dynamische Speicher Verwaltung 10

2.0.11 Klassen 10

Literaturverzeichnis **11**

Abbildungsverzeichnis **12**

Tabellenverzeichnis **13**

Quellcodeverzeichnis **14**

Abkürzungsverzeichnis **15**

A Anhang A **16**

B Anhang B **17**

1 State-of-Art Analyse

1.1 Mikrocontroller Hardware Analyse

1.1.1 Stromverbrauch und Effizienz

1.1.2 MCU Peripherie

1.1.3 Sicherheits- Features

1.2 Work-Flow und Tools in der embedded Entwicklung

1.2.1 Entwicklungs- Umgebungen

1.2.2 Test und Qualitäts- Management

1.2.3 Code Generatoren

1.2.4 Low-Level Hardware Bibliotheken

1.3 Programmiersprachen für die embedded Entwicklung

1.3.1 Kompilierte Programmiersprachen

1.3.2 Interpretierte Programmiersprachen

1.4 Verfügbare C/C++ Compiler für die ARM Architektur

1.4.1 GNU Arm Embedded Toolchain

1.4.2 IAR Embedded Workbench

1.4.3 ARM Compiler

2 Embedded C++

2.0.1 Überblick

In diesem Kapitel werden verschiedene Konzepte von C++ näher analysiert um klare Aussagen über Speicherverbrauch und Performance liefern zu können. Es soll zugleich als eine Art Guideline für Embedded Programmierer dienen um C++ effizient einsetzen zu können.

Die GNU ARM Embbedded Toolchain wird für diese Tests verwendet da Sie neben Compiler, Linker und Co. auch noch viele nützliche Tools zur Analyse mitliefert. Verschieden Tools die in den nachfolgenden Kapiteln zum Einsatz kommen werden in Tabelle 1 aufgelistet.

Tool	**Beschreibung**
arm-none-eabi-g++	C++ Compiler (Präprozessor , Kompiler, Linker)
arm-none-eabi-gcc	C Compiler (Präprozessor , Kompiler, Linker)
arm-mone-eabi-nm	Listet Symbole aus Objekt Dateien
arm-none-eabi-objdump	Zur erweiterten Analyse von Objekt Dateien
arm-none-eabi-size	Zeigt Größe der verschieden Link Segmente (Text, Data, BSS)

Tabelle 1: Analyse Tools

2.0.2 C++ Konzepte

2.0.3 Eingabe- und Ausgabestreams

2.0.4 ’auto’-Typ

2.0.5 Funktionen Überladen

C++ bietet ein Mechanismus indem man mehrere Funktionen mit dem gleichen Namen deklarieren kann, diese sich jedoch in ihrer Signatur unterscheiden müssen. Somit kann man bestehende Funktionen überladen, indem man die Datentypen oder Anzahl der Parameter verändert. Der Compiler sucht sich dann die Funktion mit der passenden Signatur heraus.

Beispiel

Im folgendem Beispiel braucht man eine Funktion die jeweils 2 **int** und **double** Werte addieren kann. Im klassischen C müsste man 2 Funktionen deklarieren die sich jeweils in ihrem Namen unterscheiden, wohingegen man in C++ den gleichen Funktions- Namen erneut verwenden darf.

```
1 int add(int a, int b){
2   return a + b;
3 }
4
5 double add(double a, double b){
6   return a + b;
7 }
8
9 int main() {
10
11   int ival_a=3, ival_b=4, iret;
12   double dval_a=3, dval_b=4, dret;
13
14   iret = add(ival_a, ival_b);
15   dret = add(dval_a, dval_b);
16
17   return 0;
18 }
```

Quellcode (1) C++ Beispiel

```
1 int add_i(int a, int b){
2   return a + b;
3 }
4
5 double add_d(double a, double b){
6   return a + b;
7 }
8
9 int main() {
10
11   int ival_a=3, ival_b=4, iret;
12   double dval_a=3, dval_b=4, dret;
13
14   iret = add_i(ival_a, ival_b);
15   dret = add_d(dval_a, dval_b);
16
17   return 0;
18 }
```

Quellcode (2) C Beispiel

Analyse

Beide Compiler generieren jeweils eine Funktion für **int** (0x30 byte) und eine für **double** (0x4c byte). Beim C++ Compiler kann man erkennen das dieser automatisch 2 Funktionen mit dem Index ’ii’ und ’dd’ erstellt. Somit kann man sagen das Funktions- Überladungen keinen Speicher Overhead erzeugen. Zudem erspart man sich lange Funktions- Namen und vermeidet dass man im Code irrtümlicherweise die falsche Funktion aufruft.

```
$ arm-none-eabi-nm.exe a.out
    --print-size
...
00008244 00000030 T _Z8addierenii
00008274 0000004c T _Z8addierendd
...
```

Compiler Output C++

```
$ arm-none-eabi-nm.exe a.out
    --print-size
...
00008244 00000030 T add_i
00008274 0000004c T add_d
...
```

Compiler Output C

Fazit

Funktions- Überladungen erzeugen keinen Speicher Overhead und sind zu empfehlen wenn sich der Funktions- Block erheblich in verschiedenen Implementationen unterscheidet. Ändern sich nur die Datentypen wie es in diesem Beispiel der Fall ist sind *Funktions- Templates* eher zu empfehlen. Ändert sich nur die Anzahl der Parameter kann man oftmals mit *Default Arguments* wie es sie in C++ gibt schöneren Code erzeugen.

2.0.6 Funktions- Templates

Oftmals muss man eine und dieselbe Funktion für verschiedene Datentypen implementieren. In C++ kann man dieses Problem mit Templates sehr elegant lösen. Templates stellen eine Art Schablone dar mit denen man Funktionen unabhängig von einem bestimmten Datentypen implementieren kann. Dies ist vor allem bei Manipulationen von Listen oder Arrays sehr interessant.

Beispiel

Das nachfolgende Beispiel zeigt wie man in C++ eine Funktion die 2 Werte addieren soll unabhängig von dessen Datentypen implementieren kann. Erkennbar ist auch dass die C++ Variante wesentlich kürzer und somit auch übersichtlicher als die Version in C wirkt.

```
1 template <typename T> T add(T a, T b) {
2   return a + b;
3 }
4
5 int main() {
6
7   int ival_a=3, ival_b=4, iret;
8   double dval_a=3, dval_b=4, dret;
9
10  iret = add(ival_a, ival_b);
11  dret = add(dval_a, dval_b);
12
13  return 0;
14 }
15
16
17
18 /*EOF*/
```

Quellcode (5) C++ Beispiel

```
1 int add_i(int a, int b){
2   return a + b;
3 }
4
5 double add_d(double a, double b){
6   return a + b;
7 }
8
9 int main() {
10
11  int ival_a=3, ival_b=4, iret;
12  double dval_a=3, dval_b=4, dret;
13
14  iret = add_i(ival_a, ival_b);
15  dret = add_d(dval_a, dval_b);
16
17  return 0;
18 }
```

Quellcode (6) C Beispiel

Analyse

Als auch schon vorigen Beispiel 2.0.5 generiert der C und C++ Compiler jeweils eine Funktion für die **int** und **double** Addition. Somit erzeugen Funktion- Templates bei richtiger Verwendung auch keinen Speicher Overhead. Erkennbar ist das der C++ Compiler die 2 Funktionen jeweils als 'WEAK' markiert. Somit wird die Template Funktion nicht als fixes Objekt deklariert und könnte von einer anderen implemetierung in einer andern Datei überschrieben werden. Um dies zu verhindern sollte sich Templates definitionen immer in einer Headerdatei befinden und wenn nötig eingebunden werden.

```
$ arm-none-eabi-nm.exe a.out
    --print-size
...
000082c8 00000030 W _Z3addIiET_S0_S0_
000082f8 0000004c W _Z3addIdET_S0_S0_
...
```

Compiler Output C++

```
$ arm-none-eabi-nm.exe a.out
    --print-size
...
00008244 00000030 T add_i
00008274 0000004c T add_d
...
```

Compiler Output C

Fazit

Funktions- Templates erhöhen die Lesbarkeit des Quellcodes und bewirken das man Fehler nur in einer Funktion beheben muss und somit eine sehr häufige Fehlerquelle ausschließen.

2.0.7 Arrays

In C++ gibt es mehrere Möglichkeiten Daten in Arrays zusammenzufassen. C- Array, *sdt::vector* und *std::array* sind die 3 bekanntesten Modelle um Daten zu speichern und werden in diesem Kapitel näher unter die Lupe genommen. Als embedded Programmierer sollte man genau wissen welche Methode für den jeweiligen Anwendungsfall die beste Wahl ist.

C- Array / std::array

Das klassische C- Array ist nach wie vor weit verbreitet und kann auch in C++ weiter verwendet werden. Jedoch sollte man in neuen Projekten darauf verzichten da es in der Standard C++ Bibliothek verschiedene Containerklassen gibt die übersichtlicher aufgebaut sind und somit gängige Fehler im Vorhinein eliminieren. Die *std::array* Containerklasse is das Pendant zum C- Array, bietet jedoch einige Vorteile was Elementzugriff oder Informationen über die Kapazität das Arrays betrifft. Näher zu erwähnen ist die Methode **at** mit der man Zugriff auf einzelne Elemente des Arrays bekommt, jedoch gegenüber **[ ]** eine Grenzen Überprüfung während der Laufzeit stattfindet. Dies kann sehr nützlich sein da der Zugriff auf nicht existierende Elemente einer der häufigsten Fehler bei C- Arrays darstellt.

Ein zusätzlicher Vorteil bieten die Standard C++ Containerklassen da diese Iteratoren zu Verfügung stellen womit man verschiedene Algorithmen der Standard Bibliothek verwenden kann. Somit kann man auf vorhandenen und getesteten Code zurückgreifen wenn es ums sortieren oder das manipulieren eines Arrays geht.

Das nachfolgende Beispiel zeigt wie man mittels C- Array und *std::array* ein **int** Array mit jeweils 10 Elementen mit 0 initialisiert und dann mit Daten befühlt.

```
1
2
3 int a[10] = { 0 };
4
5 int main() {
6
7   for(int i=0; i<10; i++){
8     a[i] = i;
9   }
10
11  return 0;
12 }
13
14 /*EOF*/
```

Quellcode (9) C- Array

```
1 #include <array>
2
3 std::array<int, 10> a = { 0 };
4
5 int main() {
6
7   for(size_t i=0; i<a.size(); ++i){
8     a[i] = i;
9   }
10
11  return 0;
12 }
13
14 /*EOF*/
```

Quellcode (10) std::array

std::vector

Bei der *sdt::vector* Containerklasse hat man die Möglichkeit die Größe des Arrays dynamisch anzupassen. Somit ist *sdt::vector* wesentlich komplexer und mit Vorsicht zu genießen da Speicher zur Laufzeit am Stack oder Heap alloziert werden muss. Bei embedded Projekten sollte man auf die *sdt::vector* Klasse verzichten und wenn eine dynamische Speicherverwaltung von Nöten ist bieten gängige RTOS oder OS Systeme bessere Mechanismen an, die im Fehlerfall (Heap/Stack Überläufe) wesentlich besser zu Debuggen sind.

Analyse

Kompiliert man die C- Array und *std::array* Variante jeweils mit der Optimierungsstufe **-0s** kann man keinen unterschied zwischen den beiden Implementierungen feststellen. Jedoch einsteht ein geringer Overheat bei Verwendung von *std::array* in Kombination mit niedrigeren Optimierungsstufen die durch die höhere Abstraktion zum eigentlichen Speicher Segment zu erklären ist. Auch zu erkennen ist dass bei beiden Implementierungen jeweils Speicher für genau 10 **int** Werte statisch in der **bss** Sektion reserviert wird.

```
$ arm-none-eabi-size a.out
text  data  bss  dec   hex
1840  1092  64   2996  bb4

$ arm-none-eabi-nm a.out --print-size
...
00018bb0 00000028 B a
...
```

C- Array

```
$ arm-none-eabi-size a.out
text  data  bss  dec   hex
1840  1092  64   2996  bb4

$ arm-none-eabi-nm a.out --print-size
...
00018bb0 00000028 B a
...
```

std::array

Fazit

Schlussfolgernd kann man Sagen dass *std::array* ohne weiteres in embedded Projekten verwenden werden darf und eine Reihe von Vorteilen gegenüber dem C- Array bietet. Auch die Kompatibilität zu älterem Code der möglicherweise noch klassische Pointer benötigt sollte kein Problem darstellen da man mit der *std::array* Methode **data** direkten Zugriff auf das darunterliegende Array bekommt. In der Standard C++ Bibliothek findet man auch noch einige andere Containerklassen wie *std::list* oder *std::deque* die sich zur Speicherung von Daten anbieten, jedoch auch wie *std::vector* wesentlich komplexer sind und somit nur zum Einsatz kommen sollten wenn Speicherverbrauch und CPU Auslastung keine wesentliche Rolle spielen.

2.0.8 Pointer vs Referenzen

2.0.9 Namespaces

2.0.10 Dynamische Speicher Verwaltung

2.0.11 Klassen

Konstruktoren

Destruktoren

Literaturverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Tabelle 1 Analyse Tools 2

Quellcodeverzeichnis

Quellcode 1 C++ Beispiel 4
Quellcode 2 C Beispiel 4
Quellcode 3 5
Quellcode 4 5
Quellcode 5 C++ Beispiel 6
Quellcode 6 C Beispiel 6
Quellcode 7 7
Quellcode 8 7
Quellcode 9 C- Array 8
Quellcode 10 std::array 8
Quellcode 11 9
Quellcode 12 9

Abkürzungsverzeichnis

ABC Alphabet

WWW world wide web

ROFL Rolling on floor laughing

A Anhang A

B Anhang B