

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 28. August 2018



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, 28. August 2018

A handwritten signature in black ink, appearing to read 'KeyF' with a stylized flourish.

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 objekt-orientierten 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	8bit vs 32bit CPU	2
1.1.2	Stromverbrauch und Effizienz	2
1.1.3	MCU Peripherie	2
1.1.4	Sicherheits- Features	2
1.2	Work-Flow und Tools in der embedded Entwicklung	2
1.2.1	Entwicklungs- Umgebungen	2
1.2.2	Test und Qualitäts- Management	2
1.2.3	Code Generatoren	2
1.2.4	Low-Level Hardware Bibliotheken	2
1.3	Programmiersprachen für die embedded Entwicklung	2
1.3.1	Kompilierte Programmiersprachen	2
1.3.2	Interpretierte Programmiersprachen	2
1.4	Verfügbare C/C++ Compiler für die ARM Architektur	2
1.4.1	GNU Arm Embedded Toolchain	2
1.4.2	IAR Embedded Workbench	2
1.4.3	ARM Compiler	2
2	Embedded C++	2
2.1	Überblick	3
2.2	Funktionen Überladen	4
2.2.1	Beispiel	4
2.2.2	Analyse	4
2.2.3	Fazit	5
2.3	Standardparameter	6
2.3.1	Beispiel	6
2.3.2	Analyse	7
2.3.3	Fazit	7
2.4	Funktions- Templates	8
2.4.1	Beispiel	8
2.4.2	Analyse	8
2.4.3	Fazit	9

2.5	Die Standardbibliothek	10
2.5.1	Überblick	10
2.5.2	Arrays	11
2.5.3	Zeichenketten	13
2.6	Zeiger vs Referenzen	19
2.6.1	Analyse	20
2.6.2	Fazit	20
2.7	Namensräume	21
2.7.1	Fazit	22
2.8	Klassen	23
2.8.1	Erstes Beispiel	23
2.8.2	RAII-Idiom	24
2.8.3	Konstruktoren	25
2.8.4	Destruktoren	27
2.8.5	Vererbung und Polymorphie	27
2.9	Operatoren Überladen	28
2.10	Dynamische Speicher Objekte	28
2.11	Fehlerbehandlung	28
2.12	Eingabe- und Ausgabestreams	28
	Literaturverzeichnis	29
	Abbildungsverzeichnis	30
	Tabellenverzeichnis	31
	Quellcodeverzeichnis	32
	Abkürzungsverzeichnis	34

1 State-of-Art Analyse

Dieses Kapitel soll einen Überblick über aktuelle Mikrocontroller Architekturen und dessen Tools zur Entwicklung darstellen. Die Auswahl des richtigen Mikrocontrollers für ein Projekt ist eines der wichtigsten Schritte was einiges an Zeit und Recherche beansprucht. Auch das auswählen der richtigen Toolchain oder die Frage ob ein RTOS oder OS zum Einsatz kommen soll wird immer wichtiger und hängt stark davon ab für welchen Mikrocontroller oder Architektur man sich entschieden hat. Stellt man während der Projektphase fest das der verwendete Mikrocontroller ungeeignet ist oder bestimmte Hardware Features fehlen, ist das meist ein großes Risiko welches den Erfolg des Projektes beeinflussen kann.

1.1 Mikrocontroller Hardware Analyse

Dieses Kapitel soll die Hardware Features und die gängigsten CPU Architekturen näher beschreiben. Der Trend neigt dazu dass immer mehr Funktionalität welche früher als externe Peripherie an den Mikrocontroller angeschlossen werden musste in diesen integriert wird. Dies führt dazu dass moderne Mikrocontroller immer komplexer werden was die Verwendung von Softwarebibliotheken welche die Hardware ansprechen immer notwendiger macht. Auch Co-degeneratoren mit denen man die komplexe Hardware über eine User Interface komfortabel konfigurieren kann finden immer mehr Einzug in die Entwicklungsumgebungen der verschiedenen Halbleiter Hersteller. Dieses Phänomen kann man gut an der Seitenanzahl vom Datenblatt gängiger Mikrocontroller erkennen. Hatten frühere 8 Bit Mikrocontroller noch 500 oder weniger Seiten findet man heute kaum noch ein Datenblatt mit weniger als 2000 Seiten. Auch die Energieeffizienz scheint eine immer wichtigere Rolle zu spielen was durch den steigenden Anteil von mobilen Geräten zu erklären ist.

1.1.1 8bit vs 32bit CPU

1.1.2 Stromverbrauch und Effizienz

1.1.3 MCU Peripherie

1.1.4 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++

2.1 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 Embedded 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 verschiedenen Link Segmente (Text, Data, BSS)

Tabelle 1: Analyse Tools

2.2 Funktionen Überladen

In C++ ist es erlaubt mehrere Funktionen mit dem gleichen Namen zu deklarieren, 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.

2.2.1 Beispiel

Im folgendem Beispiel braucht man eine Funktion die jeweils 2 *int* und *double* Werte addieren kann. Im klassischen C müsste man jeweils 2 Funktionen deklarieren die sich in ihrem Namen unterscheiden, wohingegen man in C++ den gleichen Funktionsnamen 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 i_1=3, i_2=4, i_3=0;
10 double d_1=3, d_2=4, d_3=0;
11
12 int main() {
13
14     i_3 = add(i_1, i_2);
15     d_3 = add(d_1, d_2);
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 i_1=3, i_2=4, i_3=0;
10 double d_1=3, d_2=4, d_3=0;
11
12 int main() {
13
14     i_3 = add_i(i_1, i_2);
15     d_3 = add_d(d_1, d_2);
16
17     return 0;
18 }
```

Quellcode (2) C Beispiel

2.2.2 Analyse

Beide Compiler generieren jeweils eine Funktion für *int* (48 byte) und eine für *double* (76 byte). Beim C++ Compiler kann man erkennen das dieser automatisch 2 verschiedene Funktionen mit dem Index 'ii' und 'dd' erstellt. Somit erzeugen Funktionsüberladungen keinen zusätzlichen Speicher Overhead. Auch lange Funktionsnamen können vermieden werden und die Wahrscheinlichkeit das man irrtümlicherweise die falsche Funktion verwendet sinkt erheblich.

<pre>\$ arm-none-eabi-g++ -c -O0 main.cpp \$ arm-none-eabi-nm main.o --print-size ... 00000030 0000004c t _ZL3adddd 00000000 00000030 t _ZL3addii ...</pre>	<pre>\$ arm-none-eabi-gcc -c -O0 main.c \$ arm-none-eabi-nm main.o --print-size ... 00000030 0000004c t add_d 00000000 00000030 t add_i ...</pre>
Analyse C++	Analyse C

2.2.3 Fazit

Funktionsüberladungen erzeugen keinen Speicher Overhead und sind zu empfehlen wenn sich der Funktionsblock erheblich in verschiedenen Implementierungen unterscheidet. Ändern sich nur die Datentypen wie es in diesem Beispiel der Fall ist sind *Funktionstemplates* zu bevorzugen. Ändert sich nur die Anzahl der Parameter kann man oftmals mit *Standardparametern* (Default Arguments) wie es sie in C++ gibt eleganteren Code erzeugen.

2.3 Standardparameter

Bei der Initialisierung von Hardware Modulen sind oft Standardparameter sinnvoll. Bei einer Seriellen Schnittstelle ist die Konfiguration *9600 8N1* für viele Anwendungen schon ausreichend. In C++ kann man Funktionsparameter mit festen Standardwerten versehen welche geltend sind sollte beim Funktionsaufruf das jeweilige Argument fehlen. Im klassischen C gibt es mehrere Möglichkeiten so ein Verhalten nach zu implementieren, jedoch ist dies ohne zusätzlichen Quellcode meist nicht möglich.

2.3.1 Beispiel

Das folgende Beispiel zeigt wie man die Initialisierungsfunktion einer Seriellen Schnittstelle mit Standardwerten versehen kann. Sofort erkennbar ist das die gleiche Implementierung in C wesentlich unübersichtlicher wirkt. In der main Funktion werden die verschiedenen Möglichkeiten dargestellt wie man die Funktion aufrufen könnte.

```
1 int uart_init(int baud=9600,
2               int Ndata=8,
3               int parity=0,
4               int stop=1) {
5
6     return baud+Ndata+parity+stop;
7 }
8
9
10
11
12
13
14
15
16
17
18 int ret=0;
19
20 int main() {
21
22     ret = uart_init();
23     ret += uart_init(57600);
24     ret += uart_init(57600, 7);
25     ret += uart_init(57600, 7, 1);
26     ret += uart_init(57600, 7, 1, 0);
27     return 0;
28 }
```

Quellcode (5) C++ Beispiel

```
1 int uart_init(int baud,
2               int Ndata,
3               int parity,
4               int stop) {
5
6     if(baud < 0)
7         baud = 9600;
8     if(Ndata < 0)
9         Ndata = 8;
10    if(parity < 0)
11        parity = 0;
12    if(stop < 0)
13        stop = 1;
14
15    return baud+Ndata+parity+stop;
16 }
17
18 int ret=0;
19
20 int main() {
21
22     ret = uart_init(-1, -1, -1, -1);
23     ret += uart_init(57600, -1, -1, -1);
24     ret += uart_init(57600, 7, -1, -1);
25     ret += uart_init(57600, 7, 1, -1);
26     ret += uart_init(57600, 7, 1, 0);
27     return 0;
28 }
```

Quellcode (6) C Beispiel

2.3.2 Analyse

Bei dieser Analyse ist die Größe der *main* Funktion im ROM wichtig, da in dieser die *uart_init* Funktion aufgerufen wird. Der C und C++ Compiler allozieren jeweils 240 Byte für die *main* Funktion in der *.text* Sektion. Analysiert man den generierten Assembler Code der *main* Funktion von beiden Compilern kann man keinen Unterschied feststellen. Somit ergänzt der C++ Compiler lediglich die nicht gegebenen Argumente mit den Standard Parametern während des Kompilierens. Die *uart_init* Funktion alloziert in der C Version mehr Speicher da hier gegebenenfalls die Standardparameter noch gesetzt werden müssen.

<pre>\$ arm-none-eabi-g++ -c -O0 main.cpp \$ arm-none-eabi-nm a.out --print-size ... 00000000 00000048 t _ZL9uart_initiiii 00000000 000000f0 T main 00000000 00000004 B ret ...</pre>	<pre>\$ arm-none-eabi-gcc -c -O0 main.c \$ arm-none-eabi-nm a.out --print-size ... 00000000 00000098 t uart_init 00000000 000000f0 T main 00000000 00000004 B ret ...</pre>
Analyse C++	Analyse C

Bei Verwendung von C++ Standardparametern sollte man darauf achten dass man die Parameter, wo die Wahrscheinlichkeit einer Änderung am Höchsten ist möglichst links in die Funktionsparameter-Liste packt. Bei der Seriellen Schnittelle wäre das die Baudrate da sich die restlichen Parameter nicht so häufig ändern.

2.3.3 Fazit

Standardparameter sind sehr nützlich wenn es sich um Konfigurations-Funktionen handelt da man diese übersichtlicher gestalten kann. Auch Funktions- Aufrufe mit langen Parameter Listen kann man vermeiden wenn einem die Standard Werte genügen. Einen Speicher Overhead oder Änderungen in der Laufzeit des Programms sind nicht zu befürchten.

2.4 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 interessant da man den Algorithmus nicht für jeden Datentypen erneut implementieren muss.

2.4.1 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 int i_1=3, i_2=4, i_3=0;
2 double d_1=5, d_2=6, d_3=0;
3
4 template <typename T> T add(T a, T b) {
5     return a + b;
6 }
7
8
9
10
11
12 int main() {
13
14     i_3 = add(i_1, i_2);
15     d_3 = add(d_1, d_2);
16
17     return 0;
18 }
```

Quellcode (9) C++ Beispiel

```
1 int i_1=3, i_2=4, i_3=0;
2 double d_1=5, d_2=6, d_3=0;
3
4 int add_i(int a, int b) {
5     return a + b;
6 }
7
8 double add_d(double a, double b) {
9     return a + b;
10 }
11
12 int main() {
13
14     i_3 = add_i(i_1, i_2);
15     d_3 = add_d(d_1, d_2);
16
17     return 0;
18 }
```

Quellcode (10) C Beispiel

2.4.2 Analyse

Beide Versionen wurden mit der Compiler Optimierung *-O0* kompiliert. Analysiert man beide Objekt Dateien findet man jeweils 2 Versionen von der Addier-Funktion. Bei der C Variante heißen die Funktionen genau gleich wie im Quellcode deklariert, wohingegen eine automatische Indexierung bei der C++ Variante erfolgt. In beiden Versionen werden jeweils 48 Byte für die *int* und 76 Byte für die *double* Funktion alloziert. Es kann durchaus sein dass bei anderen Optimierungsstufen die Funktionen als *inline* kompiliert werden, jedoch entsteht auch dann nie ein Unterschied zwischen der C und C++ Version.

<pre>\$ arm-none-eabi-g++ -c -O0 main.cpp \$ arm-none-eabi-nm a.out --print-size ... 00000000 0000004c W _Z3addIdET_S0_S0_ 00000000 00000030 W _Z3addIiET_S0_S0_ ...</pre>	<pre>\$ arm-none-eabi-gcc -c -O0 main.c \$ arm-none-eabi-nm a.out --print-size ... 00000030 0000004c T add_d 00000000 00000030 T add_i ...</pre>
Analyse C++	Analyse C

Zudem kann man erkennen das die Funktionen in der C Version mit *t* markiert sind und somit in die *.text* Sektion gelinkt werden. Würde man in einer anderen C Datei genau die gleiche Funktionen implementieren würde man nach dem linken 2 Versionen von genau der gleichen Funktion in der auszuführenden Datei (Executable) finden. Hierbei sind die Funktionen a und b fest an die jeweilige Objektdatei gebunden. In der C++ Version sind beide Funktionen als *WEAK* deklariert und befinden sich in einer eigenen Link Sektion. Dies ermöglicht es dem Linker Optimierungen vorzunehmen falls es mehrere Implementierungen des gleichen Templates in verschiedenen Objekt Dateien gibt. Dieses Compiler Verhalten ist auch unter dem Namen **Borland Model** bekannt.

<pre>\$ arm-none-eabi-objdump -d main.o -> .text: 00000000 <main>: ... -> .text._Z3addIhET_S0_S0_: 00000000 <_Z3addIhET_S0_S0_>: ... -> .text._Z3addIsET_S0_S0_: 00000000 <_Z3addIsET_S0_S0_>: ...</pre>	<pre>\$ arm-none-eabi-objdump -d main.o -> .text: 00000088 <main>: ... 00000000 <a>: ... 00000040 : ...</pre>
Analyse C++	Analyse C

2.4.3 Fazit

Funktionstemplates erhöhen die Lesbarkeit des Quellcodes und bewirken das man Fehler nur in einer Funktion beheben muss was die Wartbarkeit erheblich erhöht. Zudem kann sich die Verwendung von Templates positiv auf die Größe des Executables auswirken da der Linker Optimierungen vornehmen kann. Templates sollten auch wenn möglich in einer header Datei definiert werden und wenn nötig in der *.cpp* Datei eingebunden werden. Bei großen Projekten kann sich dies jedoch negativ auf die Übersetzungszeit auswirken da der Template Code mehrmals kompiliert werden muss.

2.5 Die Standardbibliothek

Dieses Kapitel widmet sich der Standardbibliothek welche ein wichtiger Bestandteil von C++ darstellt. Einige Begriffe welche im Zusammenhang mit der Bibliothek öfters genannt werden sollen hier näher beschrieben werden und auch die Frage ob sich der Einsatz in embedded Projekten lohnt soll hier beantwortet werden.

2.5.1 Überblick

Die Standardbibliothek ist ein fester Bestandteil von C++ und wird von der ISO Organisation verwaltet. Diese wurde ursprünglich unter dem Namen STL (Standard Template Library) von Hewlett-Packard entwickelt und in den C++ Standard integriert. Jedoch handelt es sich nach wie vor um 2 getrennte Bibliotheken die parallel weiterentwickelt werden. Die C++ Standardbibliothek beinhaltet nützliche Funktionen und Algorithmen welche sehr beliebt sind. Zusätzlich beinhaltet die Standardbibliothek auch die komplette C Bibliothek.

Container-Klassen

Container-Klassen sind generische Klassen welche andere Objekte und Datentypen speichern und verwalten können. Diese sind meistens als Template-Klassen implementiert wodurch man diese für die meisten Datentypen benutzen kann. Die bekanntesten Container sind *std::vector* und *std::array* welche auch in den nachfolgenden Kapiteln analysiert werden.

Iteratoren

An sich sind Iteratoren nichts anderes als intelligente Zeiger welche benutzt werden um durch die Elemente eines Container zu iterieren. Iteratoren sind reine Zugriffsobjekte und stellen die Schnittstelle für Algorithmen in C++ dar.

Algorithmen

Mit Algorithmen kann man Manipulationen an Container vornehmen. Diese sind Container unabhängig implementiert und benutzen Iteratoren um auf die Container Elemente zuzugreifen. Der *std::for_each* Algorithmus ist einer der bekanntesten mit dem man auf alle Elemente eines Containers Zugriff bekommt.

Die Standard Bibliothek in embedded Projekten

Die Bibliothek ist bekannt dafür dass diese eigenständig viel Speicher zur Laufzeit am Heap oder Stack alloziert. Zudem benötigt diese doch relativ viel Speicher womit sie für sehr kleine Mikrokontroller nicht in Frage kommt. Jedoch sollen in den nächsten Kapiteln einige interessante Elemente analysiert werden.

2.5.2 Arrays

In C++ gibt es mehrere Möglichkeiten Daten in Arrays zusammenzufassen. C-Array, `std::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 ist das Pendant zum C-Array, bietet jedoch einige Vorteile was Elementzugriff oder Informationen über die Kapazität des 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.

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 befüllt. C-Arrays und `std::array` sind beides statische Elemente wodurch die Größe beim kompilieren bereits bekannt sein muss und nachträglich nicht mehr verändert werden kann. Die Verwendung von `std::array` kann sich vor allem dann auszahlen wenn man viele Manipulation wie das Sortieren oder Ersetzen von Elementen vornimmt. In diesem Fall kann man auf getestete Algorithmen zurückgreifen welche durch die Bereitstellung von Iteratoren möglich ist.

```
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 (15) `std::array`

```
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 (16) C-Array

Analyse C-Array/std::array

Kompiliert man die C-Array und `std::array` Variante jeweils mit der Optimierungsstufe `-Os` kann man keinen Unterschied zwischen den beiden Implementierungen feststellen. Jedoch entsteht ein geringer Overhead 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.

<pre>\$ arm-none-eabi-g++ -c -Os st_array.cpp \$ 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 ...</pre>	<pre>\$ arm-none-eabi-g++ -c -Os c_array.cpp \$ 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 ...</pre>
<code>std::array</code>	<code>C-Array</code>

std::vector

Bei der `std::vector` Containerklasse hat man die Möglichkeit die Größe des Arrays dynamisch anzupassen. Somit ist `std::vector` wesentlich komplexer und mit Vorsicht zu genießen da Speicher zur Laufzeit am Stack oder Heap alloziert werden muss. Dies ist auch dadurch zu erkennen dass `sizeof(std::vector<int>)` nicht wie bei `std::array` oder dem C-Array die eigentliche Größe in Bytes zurückliefert sondern immer 12 welche lediglich zur Verwaltung von `std::vector` benötigt werden und die eigentlichen Daten am Heap gespeichert werden. Bei embedded Projekten sollte man auf die `std::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.

Fazit

Schlussfolgernd kann man sagen dass `std::array` ohne weiteres in embedded Projekten verwendet 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.

2.5.3 Zeichenketten

Bei textbasierten Schnittellen oder Dateioperationen kommt man um Zeichenketten nicht herum. In C++ kann man die Klasse `std::string` aus der C++ Standardbibliothek verwenden oder man benutzt weiterhin klassische C Strings. Der größte Vorteil von `std::string` ist dass dieser Datentyp seine exakte Größe kennt und somit unerlaubte Zugriffe außerhalb des zugewiesenen Speicher vermieden werden können. Dies ist ein bekanntes Problem bei C Strings in Kombination mit Funktionen aus der Standardbibliothek `string.h`, da man hier extrem aufpassen muss dass man nicht über die Grenzen hinaus operiert und Zeichenketten stets richtig mit `'\0'` terminiert.

Um die Funktionsweise von `std::string` besser zu verstehen analysieren wir die Größe der Datentypen `std::string` und `char`. Das die Größe von `char` 1 Byte ist sollte keine Überraschung sein, jedoch ist einem auf den ersten Blick unklar warum die Größe von `std::string` 24 Byte ist.

Datentyp	Größe mittels sizeof()
<code>std::string</code>	24
<code>char</code>	1

Tabelle 2: Größe von `std::string` und `char`

Analysiert man die `std::string` Klasse findet man heraus dass diese eine Instanziierung von der Template Klasse `std::basic_string` mit dem Datentypen `char` ist. Schaut man sich die Header-Datei dieser Klasse an kann man ein Datenschema wie im Quellcode 19 vereinfacht dargestellt erkennen. Nun weiß man auch warum `sizeof(std::string)` 24 ergibt. Die Daten setzen sich aus einem Pointer `*data` (4 byte) der auf den Datenblock zeigt, einer Variable `string_lenght` (4 byte) welche die Länge des Strings beinhaltet und einer Union (16 byte) welche als Buffer dient oder als Variable die Anzahl des allozierten Speichers am Heap beinhaltet.

```
1 char      *data;           // 4 byte
2 size_t    string_lenght;   // 4 byte
3
4 union {                     // 16 byte
5     char local_data[16];
6     size_t allocated_data;
7 }
```

Quellcode (19) Datenschema `std::string`

Solange die Zeichenkette weniger als 17 Zeichen beinhaltet, werden Daten im lokalen Datenblock von 16 Byte abgelegt auf welchen auch der Zeiger `*data` zeigt. Überschreitet die Zeichenkette die Länge von 16 Zeichen wird zur Laufzeit dynamisch Speicher am Heap reserviert wo

dann der neue String auch hin kopiert wird. Jetzt zeigt der Zeiger **data* nicht mehr auf den lokalen Buffer sondern auf den am Heap allozierten Speicher. Die Variable *allocated_data* beinhaltet nun die Größe des allozierten Speichers am Heap. Um Speicher zu sparen wurde der Buffer *local_data[16]* und die Variable *allocated_data* in eine Union verpackt da immer nur eine zur gleichen Zeit verwendet wird. Die Größe des allozierten Speichers am Heap muss mindestens so groß sein wie die Länge der eigentlichen Zeichenkette (*allocated_data* >= *string_lenght*).

Das nachfolgende Beispiel verdeutlicht wie dynamisch Speicher am Heap alloziert wird wenn man die Länge von *std::string* dynamisch verändert. Dazu wird die Zeichenkette bei jedem Schleifendurchgang um das Zeichen '=' erweitert. Die Funktionen *malloc* und *free* wurden so manipuliert das diese jeweils ein Text ausgeben wenn Speicher am Heap alloziert oder de-alloziert wird.

<pre> 1 int main() { 2 3 std::string str; 4 5 for(int i=2; i <33; i++){ 6 str.append("="); 7 printf("%02d:_%s\n", i, 8 str.c_str()); 9 } 10 11 return 0; 12 } 13 14 15 16 17 /*EOF*/ </pre>	<pre> 02: = 03: == :: 15: ===== 16: ===== * Allocate 31 bytes 17: ===== 18: ===== :: 30: ===== 31: ===== * Allocate 61 bytes * Deallocate 32: ===== 33: ===== </pre>
--	--

Quellcode (20) Dynamisches Verhalten

Ausgabe Quellcode (20)

Wie bei den C Strings ist auch bei *std::string* jede Zeichenkette mit '\0' terminiert. Fügt man das 17 Zeichen hinzu werden 31 Bytes am Heap alloziert da der interne Buffer mit 16 bytes jetzt zu klein ist. Wächst die Zeichenkette auf 32 Zeichen an werden weiter 61 Bytes alloziert wohin die neue Zeichenkette kopiert wird. Nach dem Kopier- Vorgang wird der alte Speicher von 31 Bytes wieder freigegeben. Erkennbar ist also dass der Speicher jeweils um das doppelte wächst sollte der momentane Speicher am Heap nicht mehr ausreichend sein. Die meisten modernen Compiler verfolgen eine ähnliche Strategie was das Verhalten von *std::string* angeht, jedoch kann man Unterschiede in der Größe des lokalen Buffers oder dem Algorithmus wie Speicher am Heap alloziert wird erkennen.

Die `std::string` Klasse stellt eine Reihe von Methoden zu Verfügung die das Arbeiten mit Strings sehr vereinfachen. Solche Methoden bieten eine ähnliche oder erweiterte Funktionalität wie die aus der Standard C Bibliothek `string.h` und können in diversen Dokumentationen nachgeschlagen werden. Da in kleinen embedded Systemen die Speicherverwaltung eine wichtige Rolle spielt sollen die Methoden welche Auskünfte über Kapazität und Länge der Zeichenkette geben näher analysiert werden. Grundsätzlich muss man bei der `std::string` Klasse zwischen 2 Größen unterscheiden: Der eigentlichen String Länge und des Speichers der momentan zu Verfügung steht. Tabelle 3 unterteilt die Methoden in genau diese 2 Gruppen.

String Länge	Kapazität
<code>size()</code>	<code>capacity()</code>
<code>length()</code>	<code>reserve()</code>
<code>empty()</code>	<code>shrink_to_fit()</code>
<code>resize()</code>	
<code>clear()</code>	
<code>max_size()</code>	

Tabelle 3: Überblick Methoden über Kapazität und Länge

Lediglich die Methoden `capacity()`, `reserve()` und `shrink_to_fit()` beziehen sich auf die Speicherverwaltung des Strings, bei allen anderen Funktionen geht es um die reine Länge der Zeichenkette. Da bei `std::string` Zeichen als ASCII-byte interpretiert werden liefern die Methoden `size()` und `length()` genau das gleiche Ergebnis. Das nachfolgende Beispiel soll Unterschiede noch einmal hervorheben.

Analysiert man die Ausgabe vom Quellcode 22 hat der String eine Länge von 10 Zeichen und eine Kapazität von 15 Byte nach der Initialisierung. Ruft man die Methode `reserve()` mit 500 auf, werden 500 Byte alloziert. Beim Aufruf der Methode `resize()` mit 5, wird lediglich die Zeichenkette auf eine Länge von 5 Zeichen reduziert. Die Methode `clear()` löscht alle Zeichen ändert aber nichts am alloziertem Speicher. Das Ergebnis der Methode `max_size()` ist nicht sehr aussagekräftig da dies nur ein konstanter Wert ist und **keine** Informationen über den noch verfügbaren Speicher am Heap liefert. Die Methode `shrink_to_fit()` die es ab C++11 gibt sollte den allozierten Speicher auf den kleinstmöglichen Wert begrenzen damit der eigentliche String noch hineinpasst. Dies funktioniert jedoch nur bedingt da die Methode nicht bindend ist und der Optimierungsgrad bei der Klasse selbst liegt.

<pre> 1 void print_string_info(std::string &str){ 2 3 printf("String:_%s\n", str.c_str()); 4 printf("Size()_=_%d\n", str.size()); 5 printf("Length()_=_%d\n", str.length()); 6 printf("Max_Size()_=_%d\n", str.max_size()); 7 printf("Capacity()_=_%d\n\n", str.capacity()); 8 } 9 10 int main(void) { 11 12 std::string msg("Hallo_Welt"); 13 print_string_info(msg); 14 15 msg.reserve(500); 16 print_string_info(msg); 17 18 msg.resize(5); 19 print_string_info(msg); 20 21 msg.reserve(50); 22 print_string_info(msg); 23 24 msg.clear(); 25 print_string_info(msg); 26 27 if(msg.empty()) { 28 device.printf("String_is_empty\n"); 29 }else{ 30 device.printf("String_is_not_empty\n"); 31 } 32 } </pre>	<pre> String: Hallo Welt size() = 10 length() = 10 max_size() = 2147483647 capacity() = 15 String: Hallo Welt size() = 10 length() = 10 max_size() = 2147483647 capacity() = 500 String: Hallo size() = 5 length() = 5 max_size() = 2147483647 capacity() = 500 String: Hallo size() = 5 length() = 5 max_size() = 2147483647 capacity() = 50 String: size() = 0 length() = 0 max_size() = 2147483647 capacity() = 50 String_is_empty </pre>
--	---

Quellcode (22) Kapazität von std::string

Ausgabe Quellcode (22)

Da wir jetzt einen relativ guten Eindruck über den Unterschied der Speicherverwaltung am RAM (Daten, Heap) von *std::string* gegenüber von C-Strings haben widmen wir uns jetzt dem ROM (Code) Speicherbedarf. Dieser wird hauptsächlich durch die Funktionen beeinflusst welche in der C und C++ Standardbibliothek definiert sind. Um den ROM Overhead von *std::string* herauszufinden wurde ein lauffähiges Programm was Strings manipuliert jeweils einmal mit Hilfe von C- Strings und *std::string* implementiert. Das nachfolgende Beispiel zeigt auch dass man bei *std::string* Operatoren wie + und « sinnvoll verwenden kann um zum Beispiel Strings aneinanderzuhängen.

Um keine Fehler in der Analyse zu begehen wurde das nachfolgende Beispiel kompiliert und gelinkt damit es auf einem realen Mikrocontroller lauffähig ist. Zur Analyse wurden nun die Größen der verschiedenen Link Sektionen von allen gelinkten Objekten der C und C++ Standardbibliothek und von der main.o Datei aufgelistet.

<pre> 1 #include <string> 2 3 4 int main() { 5 6 std::string str{"C++_ist_eine_"}; 7 8 str += "schoene_"; 9 str += "Sprache"; 10 int pos = str.find("schoene_"); 11 str.replace(pos, 8, "coooole_"); 12 13 return 0; 14 } 15 16 17 /*EOF*/ </pre>	<pre> 1 #include <string.h> 2 #include <stdlib.h> 3 4 int main() { 5 6 char *str = (char *) 7 malloc(sizeof(char) * 50); 8 9 strcpy((char *)str, "C++_ist_eine_"); 10 strcat((char *)str, "schoene_"); 11 strcat((char *)str, "Sprache"); 12 13 char *help = strstr(str, "schoene_"); 14 strncpy(help, "coooole_", 8); 15 16 return 0; 17 } </pre>
--	---

Quellcode (24) std::string

Quellcode (25) C- String

Da wir hier ein komplett lauffähiges Projekt mit OS analysieren ist die Größe der einzelnen Sektionen nicht sehr aussagekräftig sondern die Differenz zwischen beiden Implementationen von größerer Bedeutung. Erkennbar ist dass die Link Größen von der C Bibliothek und der main.o relativ ähnlich ausschauen. Jedoch fügt die C++ Bibliothek mit rund 8,5 KByte einen relativ großen Overhead an ROM (Code) dazu. Dieser doch recht hoher Overhead ist dadurch zu erklären dass die C++ Bibliothek hier zum ersten mal ins Projekt eingebunden wurde und somit nicht alles auf *std::string* zurückzuführen ist. Der Overhead ist daher zum größten Teil einmalig da die Initialisierungs- Routinen der C++ Bibliothek doch einen recht großen Anteil an dem Overhead haben dürften. Wie groß jetzt der Overhead von *std::string* wirklich ist kann man nicht so generell sagen, da es extrem von den Compiler Einstellungen (Flags) und der verwendeten C++ Bibliothek abhängt. Bei Verwendung einer auf ROM optimierten C++ Bibliothek wie zum Beispiel der UlibC++, die speziell für embedded Projekte entwickelt wurde, kann man durchaus noch etwas Speicher sparen.

Module	.text	.data	.bss	Module	.text	.data	.bss
[lib]\c.a	25452	2472	11	[lib]\c.a	24319	2472	89
[lib]\c++.a	8558	20	204	[lib]\c++.a	4	0	0
main.o	372	0	132	main.o	355	0	132

std::string

C- Strings

Fazit

Ob man nun `std::string` in seinem Projekt verwenden soll, hängt eher von der generellen Frage ab ob man Funktionen aus der Standard C++ Bibliothek verwenden will. Kann man mit dem doch relativ hohen Overhead leben bietet `std::string` doch einige Vorteile gegenüber klassischen C-Strings. Jedoch geht eher Gefahr mit der dynamischen Speicher Verwaltung von `std::string` daheer, da diese doch relativ komplex sein kann und durchaus zu bösen Fehlern am Heap oder Stack führen kann.

2.6 Zeiger vs Referenzen

Wenn große Datenblöcke einer Funktion übergeben werden sollen, übergibt man meist nicht den kompletten Datenblock sondern eine Referenz oder einen Zeiger auf diesen. In C++ kann man dazu Referenzen benutzen welche meist eine bessere Alternative zu den Zeigern darstellen. Zeiger sind Variablen die auf eine Adresse im Speicher zeigen, welche Daten oder Programmcode beinhalten können. Mit Referenzen kann man dies auch, jedoch können diese während der gesamten Laufzeit immer nur auf ein Datenobjekt referenzieren und können daher auch als eine Art Alias für ein anderes Objekt angesehen werden. Im Gegensatz zu Referenzen können Zeiger ihren Wert verändern und somit während der Laufzeit auf beliebig viele Datenblöcke zeigen. Bei Zeigern besteht somit immer die Möglichkeit dass dieser nicht auf einen gültigen Speicherblock zeigt und sollte daher immer auf dessen Gültigkeit überprüft werden.

Wenn es um das manipulieren von Datenblöcken geht, sieht man sehr oft noch die Verwendung von Zeigern da man hier in einer Schleife alle Elemente eines Datenblock durch iterieren kann indem man den Zeiger bei jedem Durchlauf erhöht. Das nachfolgende Beispiel zeigt wie man die Verwendung von Zeigern vermeiden kann indem man seinen Code etwas moderner gestaltet. Das Beispiel-Programm soll die Daten von jeweils 100 Personen ausgeben.

```
1#include "stdio.h"
2
3typedef struct {
4    int age;
5    int size;
6} Person;
7
8
9void print_info(Person *per, int cnt){
10
11    for(int i=0; i<cnt; i++){
12        printf("Age:_%d\n", per->age);
13        printf("Size:_%d\n", per->size);
14    }
15 }
16
17int main(void){
18
19    Person per[100];
20
21    print_info(per, 100);
22 }
```

Quellcode (28) Zeiger Beispiel

```
1#include <array>
2
3struct data {
4    int age;
5    int size;
6};
7typedef std::array<data, 100> Person;
8
9void print_info(Person &per){
10
11    for(auto &elem : per){
12        printf("Age:_%d\n", elem.age);
13        printf("Size:_%d\n", elem.size);
14    }
15 }
16
17int main(void){
18
19    Person per;
20
21    print_info(per);
22 }
```

Quellcode (29) Referenz Beispiel

Bei dem Zeiger-Beispiel wird zuerst ein neuer Datentyp *Person* mittels *typedef struct* angelegt welche die Daten einer Person beschreibt. In der main Funktion wird dann ein Array vom Type *Person* mit der Größe 100 angelegt. Die Funktion welche die Informationen aller Personen ausgibt benötigt dann einen Zeiger auf das Array und die zusätzliche Information wie viele Elemente vom Type *Person* ausgegeben werden sollen.

Bei dem Beispiel mit Referenzen wird zuerst eine einfache Struktur *data* mittels *struct* angelegt und dann ein neuer Datentyp mithilfe der Containerklasse *std::array* angelegt. Die Größe von 100 ist ein fixer Bestandteil dieses Datentyps womit man sich die Anzahl der auszugeben Elemente als Funktionsparameter sparen kann. Zudem kann man mit der *for-each* Schleife die es seit C++11 gibt eleganter und sicherer durch alle Elemente des Objektes iterieren.

2.6.1 Analyse

Die Analyse zeigt dass Referenzen keinesfalls einen Overhead an Speicher mit sich bringen. Das Beispiel mit Zeigern ist sogar um 9 Byte Größer was durch den zusätzlichen Parameter in der *print_info* Funktion zu erklären ist. Ersetzt man beim Zeiger Beispiel die Länge durch einen Konstanten Wert erzeugt der Compiler bei beiden Versionen genau das Gleiche Ergebnis.

<pre>\$ arm-none-eabi-g++ -c -Os main.cpp \$ arm-none-eabi-size main ... text data bss 163 0 0 ...</pre>	<pre>\$ arm-none-eabi-g++ -c -Os main.c \$ arm-none-eabi-size main.o ... text data bss 155 0 0 ...</pre>
--	--

Analyse Zeiger Beispiel

Analyse Referenze Beispiel

2.6.2 Fazit

Als Fazit kann man sagen dass man Referenzen gegenüber Zeigern wenn möglich bevorzugen soll und seinen Code so gestalten soll das die Verwendung von Zeigern überflüssig wird. Referenzen haben den Vorteil dass die Syntax mit dem Punkt Operator übersichtlicher wirkt, eine Referenz immer auf ein gültiges Datenobjekt zeigt und somit sicherer zu verwenden ist. Zudem bieten Referenzen gewisse Vorteile bei Kopier-Konstruktoren oder beim überladen von Operatoren.

2.7 Namensräume

Namensräume (Namespaces) sind vor allem dann nützlich wenn es um die modulare Programmierung geht. Hiermit können Namens Konflikte im Vorhinein eliminiert werden. Jeder war schon einmal in der Situation dass der Compiler einen Fehler produziert hat weil eine Funktion oder Variable schon einmal in einer anderen Datei deklariert wurde. Dies tritt vor allem bei größeren Projekten oder bei der Verwendung von externen Bibliotheken auf. In C++ kann man einzelnen Module in Namensräume verpacken um somit eine bessere Trennung von verschiedenen Code Teilen zu erlangen. Der Namensraum *std* in dem sich die Standard C++ Module befinden dürfte jedem ein Begriff sein. Das nachfolgende Beispiel zeigt wie man Namensräume richtig verwenden kann um eine sinnvolle Modularisierung zu erreichen.

```
1 #include "serial.h"
2
3 // Private Namespace
4 namespace {
5
6     typedef enum {INIT, NOT_INIT} state;
7     state is_init = NOT_INIT;
8
9     void send_byte(char byte) {
10         // Do something
11     }
12 }
13
14 namespace driver {
15
16     void serial_init(void) {
17
18         is_init = INIT;
19         // Do something
20     }
21
22     void serial_send(char *data,
23                     int len) {
24
25         if(is_init == NOT_INIT)
26             return;
27
28         for(int i=0; i<len; i++)
29             send_byte(data[i]);
30     }
31 } //namespace driver
```

Quellcode (32) serial.cpp

```
1 namespace driver {
2
3     // Init Serial
4     void serial_init(void);
5
6     // Send data
7     void serial_send(char *data,
8                     int len);
9
10 } //namespace driver
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31 /*EOF*/
```

Quellcode (33) serial.h

Das Beispiel soll demonstrieren wie man einen Treiber für eine Serielle Schnittstelle mithilfe von Namensräumen implementieren könnte. Zuerst haben wir den Namensraum *driver* deklariert der alle Hardware Treiber eines Mikrocontrollers beinhalten soll. In der Datei *serial.h* befinden sich die Funktionsprototypen welche als Schnittstelle für dieses Modul dienen. Diese müssen natürlich in den Namensraum *driver* verpackt werden. In der Datei *serial.cpp* werden nun die Funktionen *serial_init* und *serial_send* im Namensraum *driver* implementiert.

Erkennbar ist dass sich in der Datei *serial.cpp* noch eine anonymer Namensraum befindet. Anonyme Namensräume sind an dem fehlenden Namen erkennbar und dienen dazu Variablen, Funktionen oder Typen zu deklarieren welche nur innerhalb dieses Moduls (.cpp Datei) sichtbar sein sollen. Die Funktion *send_byte()* und die Variable *is_init* vom Type *state* sind somit nur in der Datei *serial.cpp* gültig. Dieses Verhalten kann man natürlich auch mit dem Schlüsselwort *static* erzeugen, jedoch mit dem Nachteil dass Typdeklarationen nicht nach außen verborgen werden. Der neue Type *state* der den Status des Moduls beschreibt macht natürlich nur Sinn in dieser .cpp Datei und sollte daher nicht nach außen sichtbar sein. Das Nachfolgende Beispiel zeigt 2 Methoden wie man das zuvor erstellte Modul verwenden kann.

```
1 #include "serial.h"
2
3 using namespace driver;
4
5 int main(void) {
6
7     serial_init();
8
9     // Do something
10    return 0;
11 }
```

Quellcode (34) using keyword

```
1 #include "serial.h"
2
3
4
5 int main(void) {
6
7     driver::serial_init();
8
9     // Do something
10    return 0;
11 }
```

Quellcode (35) :: Scope Operator

Mit dem Schlüsselwort *using* kann man dem Compiler mitteilen dass man nun die Funktionen welche sich im Namensraum *driver* befinden verwenden möchte. Möchte man jedoch nicht den ganzen Namensraum einbinden kann man mit dem Scope Operator *::* auf Funktionen oder Variablen aus dem jeweiligen Namensraum zugreifen.

2.7.1 Fazit

Namensräume sind ideal wenn es um die Modularisierung von Projekten geht und bieten sich perfekt an um Betriebssystem Funktionalitäten vom User Space zu trennen. Auch anonyme Namensräume sollten verwendet werden um Funktionalität nach außen hin zu verbergen. Ein Speicher Overhead kann auch nicht entstehen da Namensräume lediglich dazu dienen dem Compiler mitzuteilen welche Funktionen er zu verwenden hat.

2.8 Klassen

In diesem Kapitel werden die Klassen analysiert welche die Grundlage für das OOP Konzept in C++ darstellen. Dazu wird eine Beispiel Klasse implementiert die eine Serielle Schnittstelle eines Mikrocontrollers beschreiben soll. Diese Klasse wird dann fortlaufend während des Kapitels erweitert und adaptiert um zu analysieren wie sich Klassen und dessen Konzepte in Bezug auf Performance und Speicher verhalten. Des weiteren werden in diesem Kapitel einige Richtlinien aufgestellt wie man Klassen sinnvoll in embedded Projekten einsetzen kann.

2.8.1 Erstes Beispiel

Die erste Definition unserer Klasse *Serial* beinhaltet 2 private Variablen (*baudrate*, *mode*) und 2 öffentliche Methoden (*set_baudrate*, *set_mode*). Wie im Beispiel ersichtlich sollte man Membervariablen wenn Möglich immer im *private* Block definieren und den Zugriff nur über Methoden (Getters / Setters) erlauben. Dies hat den Vorteil dass man die Variablen nur über die öffentlichen Schnittstellen (*set_baudrate*, *set_mode*) verändern kann und somit auch den Wertebereich überprüfen kann. Das direkte Verbergen von Daten führt zu einer Datenkapselung womit man die Klasse als einzelnes Modul besser testen kann und man die Gewissheit hat dass die Variablen sich im gewünschten Wertebereich befinden. Die Klassendefinition und die Implementierung der Klasse sollte man auch immer in separate Dateien verpacken was zu einer besseren Übersichtlichkeit führt.

```
1 #include "serial.h"
2
3 void Serial::set_baudrate(int baudrate) {
4
5     this->baudrate = baudrate;
6 }
7
8 void Serial::set_mode(int mode) {
9
10    this->mode = mode;
11 }
```

Quellcode (36) Klassen Implementierung

```
1 class Serial {
2
3     public:
4     void set_baudrate(int baudrate);
5     void set_mode(int mode);
6
7     private:
8     int baudrate;
9     int mode;
10
11 };
```

Quellcode (37) Klassendefinition

Analysiert man mit *sizeof(Serial)* die Größe unserer Klasse erhält man das Ergebnis 8 Byte. Diese 8 Byte setzen sich aus den 2 *int* Werten zusammen welche jeweils 4 Byte allozieren. Somit benötigt eine Klasse keinen zusätzlichen Speicher neben den eigentlichen Klassen Variablen solange man auf Virtuelle Methoden und Vererbung verzichten kann. Für diesen Fall benötigt eine C++ Klasse nicht mehr Speicher als eine äquivalente C Struktur. Lediglich eine leere Klasse alloziert 1 Byte im RAM Speicher.

Das nachfolgende Beispiel zeigt den Zusammenhang zwischen C Strukturen und Klassen da diese oft miteinander verglichen werden. Bis auf den großen Unterschied das man bei Klassen mit den Schlüsselwörtern *private*, *public* und *protected* die Zugriffskontrolle einer Klasse steuern kann ähneln sich beide sehr.

```
1 class Base{
2     int id;
3 };
4
5 class Serial : public Base {
6     int baudrate;
7     int mode;
8 };
9 /*EOF*/
```

Quellcode (38) Klasse C++

```
1 struct Base{
2     int id;
3 };
4
5 struct Serial{
6     struct Base base;
7     int baudrate;
8     int mode;
9 };
```

Quellcode (39) C Structur

Wie im Beispiel ersichtlich kann man mit C Strukturen sogar eine Art Vererbung nach implementieren indem man einfach mehrere Strukturen ineinander verschachtelt.

2.8.2 RAI-Idiom

Bevor wir mit den Konstruktoren und Destruktoren fortfahren soll hier kurz das RAI-Idiom erläutert werden welches eine wichtige Programmieretechnik in C++ darstellt. RAI steht für "Resource acquisition is initialization" was dafür sorgen soll dass bei der Ressourcenbelegung für ein Objekt die Initialisierung für dieses gleich mit erfolgt. Das nachfolgende Beispiel zeigt wie eine Array RAI oder nicht RAI Konform angelegen werden kann. Bei der *Vector* Containerklasse wird automatisch der Konstruktor beim anlegen des Objektes aufgerufen und der Destruktor wenn der Funktionsblock verlassen wird. Bei dem nicht RAI Konformen Beispiele wo Speicher mit *new* alloziert wurde muss man das Array selber noch mit Initialisierungs-Daten befüllen und nach der Verwendung auch wieder dafür sorgen dass der allozierte Speicher mit *delete* freigegeben wird.

```
1 void func(void)
2 {
3     vector<char> buffer = {'A', 'B'};
4
5     // buffer verwenden
6 }
7
8
9 /*EOF*/
```

Quellcode (40) RAI Konform

```
1 void func(void)
2 {
3     char* buffer = new char[2];
4     buffer[0] = 'A';
5     buffer[1] = 'B';
6
7     // buffer verwenden
8     delete[] buffer;
9 }
```

Quellcode (41) Nicht RAI Konform

Dieses Konzept ist eines der größten Vorteile von C++ da hier der Compiler automatisch dafür sorgt dass beim anlegen eines neuen Objektes am Stack oder Heap der Konstruktor aufgerufen wird. Auch der Destruktor wird automatisch ausgeführt sollte der Block indem das Objekt angelegt wurde verlassen werden. Das automatische aufrufen von Konstruktor und Destruktor kann das Laufzeitverhalten des Programms natürlich extrem beeinflussen und kann für unerfahrene C++ Programmierer eine Fehlerquelle darstellen da es nicht immer ersichtlich ist wann diese aufgerufen werden. Das erzeugen temporärer Objekte in Schleifen ist ein häufiger Grund warum C++ Programme langsam werden.

2.8.3 Konstruktoren

Um die Konstruktoren näher zu analysieren erweitern wir unsere *Serial* Klasse um 2 Konstruktor Methoden. Der Standard Konstruktor welcher geltend ist wenn bei der Definition des Objektes keine Initialisierungswerte angegeben wurden. Diesen sollte man immer implementieren um das Objekt mit Standard Werten zu initialisieren außer man will explizit erreichen dass man das Objekt nur mit einem benutzerdefinierten Konstruktor erstellen kann. Mit dem 2 Konstruktor kann man das Objekt mit benutzerdefinierten Werten für *baudrate* und *mode* anlegen. Bei den Konstruktoren gelten die gleiche Regeln wie bei den Funktionsüberladungen womit man theoretisch eine unbegrenzte Zahl an Konstruktoren definieren kann. Zudem können auch Standardwerte in Konstruktoren verwendet werden.

<pre> 1 #include "serial.h" 2 3 Serial::Serial() 4 : baudrate{115000}, mode{2} { } 5 6 Serial::Serial(int _baud, int _mode) 7 : baudrate{_baud}, mode{_mode} { } 8 9 void Serial::set_baudrate(int baudrate) { 10 11 this->baudrate = baudrate; 12 } 13 14 void Serial::set_mode(int mode) { 15 16 this->mode = mode; 17 } </pre>	<pre> 1 class Serial { 2 3 public: 4 // ctors 5 Serial(); 6 Serial(int _baud, int _mode); 7 8 void set_baudrate(int baudrate); 9 void set_mode(int mode); 10 11 private: 12 int baudrate; 13 int mode; 14 15 }; 16 17 /*EOF*/ </pre>
---	--

Quellcode (42) Klassen Implementierung

Quellcode (43) Klassendefinition

Bei der Definition globaler Objekte muss man sich zusätzlich Gedanken machen zu welchem Zeitpunkt der Konstruktor aufgerufen wird und ob dies zu diesem Zeitpunkt schon sinnvoll ist. Konstruktoren von globalen Objekten werden vor dem Eintritt in die *main* Funktion aufgerufen

wodurch das System möglicherweise noch nicht zu 100 Prozent initialisiert ist. Das nachfolgende Beispiel soll dieses Problem verdeutlichen.

```
1 #include "serial"
2
3 // Create Serial Object global
4 Serial com(9600, 2);
5
6 int main(void) {
7
8     // Init System
9     SystemInit();
10
11     // Send Byte
12     com.sendByte(0xAA); //Error
13
14     return 0;
15 }
```

Quellcode 44: Konstruktor Aufruf bei globalen Objekten

Üblicherweise müssen Mikrocontroller nach dem Power-On Reset initialisiert werden damit das Bus-System oder Taktsignal ordnungsgemäß funktioniert. Dies wird üblicherweise als erstes in der *main* Funktion erledigt. Würde man jetzt im Konstruktor Operationen durchführen welche eine korrekte Initialisierung des Systems voraussetzen würde es zu einem Fehler kommen. In diesem Beispiel würde die Initialisierung der Seriellen Schnittelle im Konstruktor fehlschlagen da das Peripherie-Modul noch über kein Taktsignal verfügt. Um dieses Problem zu umgehen kann man eine separate Methode zur Initialisierung bereitstellen welche dann nach der Systeminitialisierung aufgerufen werden muss, oder man müsste im Konstruktor überprüfen ob das System bereits initialisiert wurde und wenn nicht eine Systeminitialisierung durchführen. Die eleganteste Methode ist jedoch wenn man das Objekt nicht global anlegt sondern erst dynamisch nach der Systeminitialisierung erzeugt. Verwendet man jedoch ein RTOS oder anderes OS System kann man davon ausgehen dass die Systeminitialisierung bereits durchgeführt wurde bevor Konstruktoren von globalen Objekten aufgerufen werden.

Wir zur Laufzeit eine Instanz eines Objektes erzeugt wird dieses üblicherweise am Stack angelegt und wieder entfernt sobald der Gültigkeitsbereich in welcher die Instanz erstellt wurde verlassen wird. Solange das jeweilige Objekt über normale Datentypen wie (*int*, *double*, ...) enthält funktioniert das automatisch. Wurde jedoch beim anlegen der Instanz Speicher mit *new*, *malloc* im Konstruktor alloziert muss man sich selber im Destruktor darum kümmern das dieser wieder ordnungsgemäß freigegeben wird. Destrukturen und der Lebensbereich von Instanzen werden im nächsten Kapitel näher analysiert.

2.8.4 Destruktoren

2.8.5 Vererbung und Polymorphie

2.9 Operatoren Überladen

2.10 Dynamische Speicher Objekte

2.11 Fehlerbehandlung

2.12 Eingabe- und Ausgabestreams

Literaturverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Tabelle 1	Analyse Tools	3
Tabelle 2	Größe von std::string und char	13
Tabelle 3	Überblick Methoden über Kapazität und Länge	15

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++ Beispiel	8
Quellcode 10	C Beispiel	8
Quellcode 11	9
Quellcode 12	9
Quellcode 13	9
Quellcode 14	9
Quellcode 15	std::array	11
Quellcode 16	C-Array	11
Quellcode 17	12
Quellcode 18	12
Quellcode 19	Datenschema std:string	13
Quellcode 20	Dynamisches Verhalten	14
Quellcode 21	14
Quellcode 22	Kapazität von std:string	16
Quellcode 23	16
Quellcode 24	std:string	17
Quellcode 25	C- String	17
Quellcode 26	17
Quellcode 27	17
Quellcode 28	Zeiger Beispiel	19
Quellcode 29	Referenz Beispiel	19
Quellcode 30	20
Quellcode 31	20
Quellcode 32	serial.cpp	21
Quellcode 33	serial.h	21
Quellcode 34	using keyword	22

Quellcode 35 :: Scope Operator	22
Quellcode 36 Klassen Implementierung	23
Quellcode 37 Klassendefinition	23
Quellcode 38 Klasse C++	24
Quellcode 39 C Structur	24
Quellcode 40 RAI Konform	24
Quellcode 41 Nicht RAI Konform	24
Quellcode 42 Klassen Implementierung	25
Quellcode 43 Klassendefinition	25
Quellcode 44 Konstruktor Aufruf bei globalen Objekten	26

Abkürzungsverzeichnis

STL	Standard Template Library
ISO	International Organization for Standardization
ROM	read-only memory
RAM	random-access memory
RTOS	Real Time Operating System
OS	Operating System