MASTERARBEIT

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

Objektorientierung in Embedded Systems

Ausgeführt von: Ney Fränz, BSc

Personenkennzeichen: 1610297013

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

Wien, den 23. September 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, 23. September 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

Abstract

Much has happened in the development of embedded systems in recent years. Modern programming languages such as C/C ++ have established themselves in the embedded development and complex programming in assembler should only be necessary in a few cases. This thesis mainly deals with the question of whether the use of an object-oriented programming language on platforms with only a few kbytes of flash memory makes sense and what advantages this brings to the development of embedded systems.

In particular, the most common concepts (classes, templates, etc.) of the object-oriented language should be analyzed in order to make a statement about performance and memory consumption. For this purpose, the compiled code is analyzed and various benchmark tests are performed. In addition, the comparison with a classical functional programming language is presented.

C/C++ is used as reference programming language in conjunction with the ARM Cortex-M architecture. This combination is very interesting for small and low-power IoT projects and is likely to prevail in the future. At the beginning, a state-of-the-art analysis is performed to list all available C/C++ compilers and to check which C/C++ extensions and versions are supported.

Keywords: Object-oriented programming, C/C++, Embedded Software, ARM Cortex

Inhaltsverzeichnis

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 welche 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 Codegeneratoren mit denen man die komplexe Hardware über ein 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 steigernden Anteil von mobilen Geräten zu erklären ist.

2 Embedded C++

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.

2.1 Einleitung

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

Tool	**Beschreibung**
arm-none-eabi-g++	C++ Compiler (Präprozessor, Compiler, Linker)
arm-none-eabi-gcc	C Compiler (Präprozessor, Compiler, 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 Sektionen (Text, Data, BSS)

Tabelle 1: Analyse Tools

Die Resultate über Performance und Speicher Overhead welche in den nachfolgenden Kapiteln ermittelt werden sind keine in Stein gemeißelten Werte und sollen nur einen ungefähren Überblick über den zu erwarteten Overhead geben. Die Ergebnisse können je nach der verwendeten Toolchain oder CPU etwas variieren, sollten jedoch für die meisten Anwendungsfälle gültig sein.

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.

```
$ arm-none-eabi-g++ -c -O0 main.cpp
$ arm-none-eabi-nm main.o --print-size
...
00000030 0000004c t _ZL3adddd
00000000 00000030 t _ZL3addii
...
```

Analyse C++

```
$ 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
...
```

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 Schnittelle 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 verschieden 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 gegeben 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.

```
$ 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
...
```

Analyse C++

```
$ 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
...
```

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 Funktionsaufrufe 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.

```
$ 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_
...
```

Analyse C++

```
$ 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
...
```

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 verschieden Objekt Dateien gibt. Dieses Compiler Verhalten ist auch unter dem Namen **Borland Model** bekannt.

```
$ 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_>:
...
```

Analyse C++

```
$ arm-none-eabi-objdump -d main.o

-> .text:
00000088 <main>:
...

00000000 <a>:
...

00000040 <b>:
...
```

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 is 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 bekannteste 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 Containers zu iterieren. Iteratoren sind reine Zugriffsobjekte und stellen die Schnittstelle für Algorithmen in C++ dar.

Algorithmen

Mit Algorithmen kann man Manipulationen an Containern 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 Microkontroller 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, *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 dem Zugriff mittels [ ] 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 befühlt. C-Arrays und *std::array* sind beides statische Elemente wodurch dir 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 Manipulationen 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<sizeof(a); 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 *-0s* 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.

```
$ 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
...
```

std::array

```
$ 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
...
```

C-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. 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 Byte welche lediglich zur Verwaltung von *std::vector* benötigt werden und die eigentlichen Daten am Heap gespeichert werden. 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.

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.

2.5.3 Zeichenketten

Bei textbasierten Schnittstellen 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 Speichers 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()**
std::string	24
char	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_length* (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_length;   // 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_length*).

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 einen Text ausgeben wenn Speicher am Heap alloziert oder dealloziert wird.

```
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*/
```

Quellcode (20) Dynamisches Verhalten

```
02: =
03: ==
:::
15: ==============
16: ===============
* Allocate 31 bytes
17: ================
18: =================
:::
30: ==============================
31: ===============================
* Allocate 61 bytes
* Deallocate
32: ================================
33: =================================
```

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 weitere 61 Bytes alloziert wohin die neue Zeichenkette kopiert wird. Nach dem Kopiervorgang 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 vom 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**
size()	capacity()
length()	reserve()
empty()	shrink_to_fit()
resize()	
clear()	
max_size()	

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.

```
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 }
```

Quellcode (22) Kapazität von std::string

```
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
```

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 compiliert und gelinkt damit es auf einem reellen 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.

```cpp
#include <string>


int main() {

  std::string str{"C++ ist eine "};

  str += "schoene ";
  str += "Sprache";
  int pos = str.find("schoene ");
  str.replace(pos, 8, "coooole ");

  return 0;
}


/*EOF*/
```

Quellcode (24) std::string

```cpp
#include <string.h>
#include <stdlib.h>

int main() {

  char *str = (char *)
              malloc(sizeof(char)*50);

  strcpy((char *)str, "C++ ist eine ");
  strcat((char *)str, "schoene ");
  strcat((char *)str, "Sprache");

  char *help = strstr(str, "schoene ");
  strncpy (help, "coooole ", 8);

  return 0;
}
```

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
[lib]\c.a	25452	2472	11
[lib]\c++.a	8558	20	204
main.o	372	0	132

std::string

Module	.text	.data	.bss
[lib]\c.a	24319	2472	89
[lib]\c++.a	4	0	0
main.o	355	0	132

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* einher, 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 daheer 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.

```c
#include "stdio.h"

typedef struct {
  int age;
  int size;
} Person;


void print_info(Person *per, int cnt){

  for(int i=0; i<cnt; i++){
    printf("Age: %d\n", per->age);
    printf("Size: %d\n", per->size);
  }
}

int main(void){

  Person per[100];

  print_info(per, 100);
}
```

Quellcode (28) Zeiger Beispiel

```c
#include <array>

struct data {
  int age;
  int size;
};
typedef std::array<data, 100> Person;

void print_info(Person &per){

  for(auto &elem : per){
    printf("Age: %d\n", elem.age);
    printf("Size: %d\n", elem.size);
  }
}

int main(void){

  Person per;

  print_info(per);
}
```

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 Typ *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 Typ *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.

```
$ arm-none-eabi-g++ -c -Os main.cpp
$ arm-none-eabi-size main.o
...
text    data     bss
163        0       0
...
```

Analyse Zeiger Beispiel

```
$ arm-none-eabi-g++ -c -Os main.c
$ arm-none-eabi-size main.o
...
text    data     bss
155        0       0
...
```

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 Namenskonflikte 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 verschieden 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.

```cpp
#include "serial.h"

// Private Namespace
namespace {

  typedef enum {INIT, NOT_INIT} state;
  state is_init = NOT_INIT;

  void send_byte(char byte){
    // Do something
  }
}

namespace driver {

  void serial_init(void){

    is_init = INIT;
    // Do something
  }

  void serial_send(char *data,
                   int len){

    if(is_init == NOT_INIT)
      return;

    for(int i=0; i<len; i++)
      send_byte(data[i]);
  }
} //namespace driver
```

Quellcode (32) serial.cpp

```cpp
namespace driver {

  // Init Serial
  void serial_init(void);

  // Send data
  void serial_send(char *data,
                   int len);

} //namespace driver


/*EOF*/
```

Quellcode (33) serial.h

Das Beispiel soll demonstrieren wie man einen Treiber für eine Serielle Schnittelle 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 Schnittelle 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 Typ *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 Typ *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 keyworld

```
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 Schnittelle 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 Schnittellen *(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 gewünschtem 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 baudate){
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 baudate);
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 RAII-Idom

Bevor wir mit den Konstruktoren und Destruktoren fortfahren soll hier kurz das RAII-Idom erläutert werden welches eine wichtige Programmiertechnik in C++ darstellt. RAII 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 ein Array RAII oder nicht RAII Konform angelegt 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 RAII Konformen Beispiel 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* frei gegeben wird.

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

Quellcode (40) RAII 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 RAII 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 Destruktur wird automatisch ausgeführt sollte der Block in dem 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 zweiten Konstruktor kann man das Objekt mit benutzerdefinierten Werten für *baudrate* und *mode* anlegen. Bei den Konstruktoren gelten die gleichen Regeln wie bei den Funktionsüberladungen womit man theoretisch eine unbegrenzte Zahl an Konstruktoren definieren kann. Zudem können auch Standardwerte in Konstruktoren verwendet werden.

```cpp
#include "serial.h"

Serial::Serial()
: baudrate{115000}, mode{2} { }

Serial::Serial(int _baud, int _mode)
: baudrate{_baud}, mode{_mode} { }

void Serial::set_baudrate(int baudate){

  this->baudrate = baudrate;
}

void Serial::set_mode(int mode){

  this->mode = mode;
}
```

Quellcode (42) Klassen Implementierung

```cpp
class Serial {

  public:
  // ctors
  Serial();
  Serial(int _baud, int _mode);

  void set_baudrate(int baudate);
  void set_mode(int mode);

  private:
  int baudrate;
  int mode;

};

/*EOF*/
```

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 Objeckten

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 voraussetzten 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.

Wird 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. Destruktoren und der Lebensbereich von Instanzen werden im nächsten Kapitel näher analysiert.

2.8.4 Destruktoren

Unsere Serial Klasse wird nun um einen Sende Buffer welcher bis zu 100 Zeichen speichern kann erweitert. Dieses *char* Array wird im Konstruktor mit *new* dynamisch angelegt. Anders als bei den Standard Datentypen wie *int* oder *double* wird dieser Speicher nicht automatisch wieder aufgeräumt. Verwendet man die Schlüsselwörter *new* oder *malloc* im Konstruktor muss man unbedingt daran denken einen Destruktor zu definieren wo man den allozierten Speicher wieder frei gibt. Das nachfolgende Beispiel demonstriert die Verwendung eines Destruktors im Bezug auf unsere Serial Klasse.

```
1 #include "serial.h"
2
3 Serial::Serial(int _baud, int _mode)
4 : baudrate{_baud}, mode{_mode} {
5
6     buffer = new char[100];
7 }
8
9 Serial::~Serial(){
10
11     delete[] buffer;
12 }
13
14 void Serial::set_baudrate(int baudate){
15
16   this->baudrate = baudrate;
17 }
18
19 void Serial::set_mode(int mode){
20
21   this->mode = mode;
22 }
```

Quellcode (45) Klassen Implementierung

```
1 class Serial {
2
3   public:
4   // ctors
5   Serial():Serial(9600, 1){};
6   Serial(int _baud, int _mode);
7   ~Serial();
8
9   void set_baudrate(int baudate);
10  void set_mode(int mode);
11
12  private:
13  int baudrate;
14  int mode;
15  char *buffer;
16
17 };
18
19
20
21
22 /*EOF*/
```

Quellcode (46) Klassendefinition

Zusätzlich hat unsere Serial Klasse eine Konstruktor Definition weniger als noch in der vorherigen Version. Dies ist möglich da man in C++ Konstruktoren delegieren kann was einiges an Tipparbeit ersparen kann da man nur Code für einen Konstruktor schreiben muss. Der Standardkonstruktor welcher in der vorherigen Version die Klasse mit den Default Werten *9600, 1* initialisiert hat wurde jetzt nicht mehr extra definiert sondern wird auf einen andern Konstruktor delegiert. Wenn man mehrere Konstruktoren in der Klassendefinition deklariert jedoch nur einen Konstruktor definiert auf welchen alle anderen delegiert werden ist ein sehr schöner Programmierstil da man sich nicht darum kümmern muss Fehler in jeder Konstruktor Version zu beheben. Hätte man in diesem Beispiel die Containerkasse *std::array* verwendet müsste man den Speicher nicht im Destruktor freigeben da diese Klasse schon über einen eignen Destruktor verfügt welcher automatisch aufgerufen wird.

2.8.5 Vererbung und Polymorphie

In diesem Kapitel wird der Overhead welcher bei der Verwendung von virtuellen Methoden oder mehrfacher Vererbung auftritt näher analysiert. In der Literatur liest man häufig von einem großen Overhead wenn es um virtuelle Funkionen oder mehrfache Vererbung geht, jedoch wird meist der Grund dafür nicht sehr eindeutig erläutert. Um die Probleme zu verdeutlichen werden wir zuerst eine einfache Basis Klasse kreieren von welcher dann unsere Serial Klasse abgeleitet wird.

```
1 class Com {
2
3   public:
4   void set_status(int status){
5     this->status = status;
6   };
7
8   int get_status(void){
9     return status;
10  };
11
12  virtual void send_byte(char byte){
13      printf("Com:␣%c␣\n", byte);
14  }
15
16  private:
17  int status;
18 };
```

Quellcode (47) Basis Klasse

```
1 #include "com.h"
2
3 class Serial : public Com{
4
5   public:
6   // ctors
7   Serial():Serial(9600, 1){};
8   Serial(int _baud, int _mode);
9   ~Serial();
10
11  void set_baudrate(int baudate);
12  void set_mode(int mode);
13
14  private:
15  int baudrate;
16  int mode;
17  char *buffer;
18 };
```

Quellcode (48) Klassendefinition

Wie im Beispiel ersichtlich wurde eine Klasse *Com* erzeugt welche als Basis für jegliche Kommunikationsschnittellen wie UART, SPI oder I2C dienen soll. Die Klasse verfügt über eine private Variable *status* welche über die öffentlichen Schnittellen *set_status* und *get_status* angegriffen werden kann und über eine virtuelle Methode *send_byte* welche im Normalfall von der ableitenden Klasse überschrieben werden soll. Wird diese nicht überschrieben oder verwendet man die *Com* Klasse als einzelnes Modul wird die Ausgabe von *send_byte* zum Debug Interface oder zu STDIO weitergeleitet.

Die Verwendung von Basisklassen führt meist zu sehr elegantem Code da man gemeinsame Funktionalitäten wie das auch bei Kommunikationsschnittstellen der Fall ist sehr gut gruppieren kann. Auch für Benutzer einer API bringt das enorme Vorteile da die Funktionen einer SPI oder I2C Schnittelle sich in vielen Punkten ähneln was zu einer besseren Übersichtlichkeit führt. Jedoch muss man viel mehr Zeit in die Design Phase investieren und sich genau überlegen wie man die Funktionalität sinnvoll modularisieren kann.

Einfache Vererbung

Zuerst analysieren wir die einfache Vererbung ohne Verwendung der virtuellen Methode *send_byte*. In diesem Fall wird die Basisklasse *Com* lediglich um die Funktionalität welche in der *Serial* Klasse definiert wurde erweitert. Man spricht dann von einfacher Vererbung wenn eine Klasse immer nur von einer anderen Klasse erbt. Das nachfolgende Diagramm soll dies verdeutlichen.

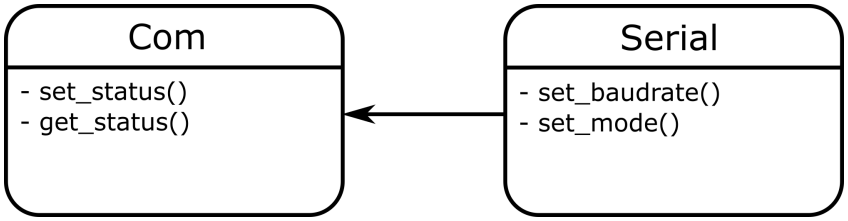

Abbildung 25: Einfache Vererbung

Die Verwendung von einfacher Vererbung erzeugt keiner Overhead an Speicher und verändert auch das Laufzeitverhalten des Programmes nicht. Dies ist möglich da der Compiler die Funktionen statisch Binden kann da es immer eindeutig ist welche Funktion aufgerufen werden soll. Um zu analysieren ob der Compiler keine zusätzlichen Daten einbaut überprüfen wir die Größe der Serial Klasse mit *sizeof*. Das Ergebnis von sizeof(Serial) liefert 16 Byte welche sich wie in Tabelle 4 beschrieben zusammensetzen.

Variable	**Bytes**
int baudrate (Serial)	4
int mode (Serial)	4
char *buffer (Serial)	4
int status (Com)	4
sizeof(Serial)	16

Tabelle 4: Sizeof(Serial)

Die 16 Byte setzen sich nur aus den Member Variablen zusammen welche in den Klassen *Com* und *Serial* definiert wurden. Somit ist bewiesen das einfache Vererbung keinen Speicher Overhead mit sich bringt und sich identisch verhält als hätte man in C mehrere Strukturen ineinander verschachtelt. Bei der einfachen Vererbung werden die Elemente der Basisklasse einfach vor die Elemente der abgeleiteten Klasse gelegt und der Aufruf einer Elementfunktion erfolgt auch ohne zusätzlichen Overhead.

Virtuelle Methoden

Virtuelle Methoden erlauben es einer Klasse Methoden zu überschreiben welche bereits (virtuell) in der Basisklasse definiert wurden. Jedoch führt die Verwendung von virtuellen Methoden zu einigen Schwierigkeiten da der Compiler die Methoden jetzt nicht mehr statisch Binden kann sondern zur Laufzeit ein dynamisches Binden der Methoden durchgeführt werden muss. Um dieses Problem zu lösen fügt der Compiler eine Tabelle (Virtual Function Table) ein welche die Referenz auf die zu verwendete Methode beinhaltet. Diese Tabelle wird im Konstruktor des Objektes erzeugt und muss jedesmal durchsucht werden, wenn eine virtuelle Funktion aufgerufen wird. Dies hat den Nachtteil, dass die Tabelle zusätzlichen Speicher belegt und das Laufzeitverhalten beeinflusst wird da bei jedem Aufruf einer virtuellen Funktion zusätzlicher Code ausgeführt werden muss.
Um dieses Problem zu verdeutlichen erweitern wir unsere Serial Klasse so, dass die virtuellen Funktionen, welche in der Basisklasse *Com* virtuell implementiert wurden, überschrieben werden. Seit C++11 kann man mit dem Schlüsselwort *override* dem Compiler explizit sagen, dass eine Funktion überschrieben werden soll was bei schwer auffindbaren Tippfehlern hilfreich sein kann.

```
1 #include "serial.h"
2
3 Serial::Serial(int _baud, int _mode)
4 : baudrate{_baud}, mode{_mode} {
5
6     buffer = new char[100];
7 }
8
9 Serial::~Serial(){
10
11     delete[] buffer;
12 }
13
14 void Serial::set_baudrate(int baudate){
15
16   this->baudrate = baudrate;
17 }
18
19 void Serial::set_mode(int mode){
20
21   this->mode = mode;
22 }
23
24 void Serial::send_byte(char byte){
25   printf("Serial:␣%c␣\n", byte);
26 }
```

Quellcode (49) Klassen Implementierung

```
1 #include "com.h"
2
3 class Serial : public Com{
4
5   public:
6   // ctors
7   Serial():Serial(9600, 1){};
8   Serial(int _baud, int _mode);
9   ~Serial();
10
11  void set_baudrate(int baudate);
12  void set_mode(int mode);
13  void send_byte(char byte);
14
15  private:
16  int baudrate;
17  int mode;
18  char *buffer;
19 };
20
21
22
23
24
25
26 /*EOF*/
```

Quellcode (50) Klassendefinition

Wie in Abbildung 27 ersichtlich besteht nun das Problem das man zur Laufzeit eine dynamische Bindung der Funktion *send_byte* mit Hilfe der *Virtual Function Table* durchführen muss.

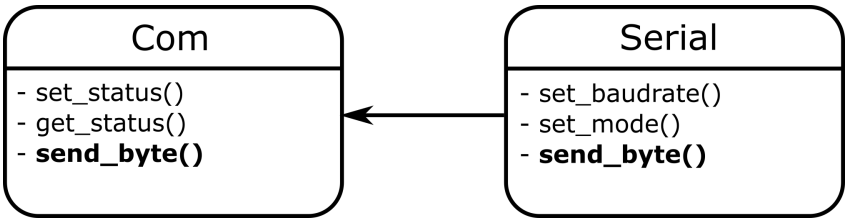

Abbildung 27: Vererbung mit virtuellen Methoden

Um nun zu Überprüfen wie sich virtuelle Funktionen auf den Speicherverbrauch unserer *Serial* Klasse auswirken schauen wir uns das Ergebnis von *sizeof(Serial)* erneut an. Nun erhält man das Ergebnis von 20 Byte welche sich wie in Tabelle 5 beschrieben zusammensetzen. Ein zusätzlicher Zeiger (4 Byte) welcher auf die *Virtual Function Table* zeigt ist in der Basisklasse *Com* dazugekommen. Die VFT kann je nach Compiler Konfiguration in eine Sektion im ROM oder RAM gelinkt werden und beinhaltet für jede virtuelle Funktion einen Eintrag. Somit entsteht ein Speicher Overhead, welcher sich durch die Größe der VFT Tabelle und dem Zeiger auf diesen zusammensetzt.

Variable	**Bytes**
int baudrate (Serial)	4
int mode (Serial)	4
char *buffer (Serial)	4
int status (Com)	4
Pointer to VFT (Com)	4
sizeof(Serial)	20

Tabelle 5: Sizeof(Serial)

Die Verwendung von vielen virtuellen Funktionen wirkt sich dadurch extrem negativ auf die Laufzeit des Programmes aus da bei jedem Funktionsaufruf einer virtuellen Funktion die Tabelle durchsucht werden muss und diese auch beim Anlegen eines solchen Objektes zuerst erzeugt werden muss. Der genaue Ovehead hängt natürlich von der Größe der VFT Tabelle ab.

Ein Beispiel soll noch einmal verdeutlichen warum die VFT Tabelle benötigt wird und warum der Compiler keine statische Bindung der Funktion *send_byte* zur Übersetzungszeit durchführen kann. Im nachfolgenden Beispiel wird ein Objekt *ser* vom Typ *Serial* angelegt. Zusätzlich wurde eine Funktion *send* implementiert die ein Zeichen auf der verwendeten Schnittelle ausgibt. Die Funktion benötigt dafür einen Parameter welcher vom Typ *Com* ist um die Funktion allgemein zu halten. Somit kann die Funktion für alle Objekte verwendet werden welche von

der Basisklasse *Com* erben und ist unabhängig von der eigentlichen Schnittelle (I2C, SPI oder UART). Darin besteht auch jetzt das Problem da die Funktion *send* ohne Hilfe der VFT nicht weiß welche *send_byte* Funktion verwendet werden soll. Dies ist auch der Grund warum sich der Zeiger welcher auf die VFT Tabelle zeigt auch immer in der Basisklasse befinden muss was wir vorher mit *sizeof(Serial)* auch herausgefunden haben.

```
1 #include "com.h"
2 #include "serial.h"
3
4 void send(Com& arg) {
5   arg.send_byte('A');
6 }
7
8 int main(void){
9
10   Serial ser;
11
12   send(ser);
13
14   return 1;
15 }
```

Quellcode 51: Beispiel Virtuelle Funktion

Mehrfache Vererbung

Bei einer mehrfachen Vererbung erbt eine Klasse von mehr als einer Basisklasse. Abbildung 28 zeigt ein mögliches Konstrukt einer mehrfachen Vererbung. Gegenüber anderen objektorientierten Sprachen ist eine mehrfache Vererbung in C++ erlaubt, sollte jedoch mit Vorsicht verwendet werden weil man sich seine Modularisierung komplett zerstören kann wenn man nicht richtig aufpasst. Der Overhead hält sich noch relativ in Grenzen da der *this* Zeiger gelegentlich angepasst werden muss wenn eine Methode aus einer Basisklasse aufgerufen wird. Der Compiler erzeugt eine sogenannte *thunk* Methode welche den *this* Zeiger adaptiert. Noch komplizierter wird es wenn die Basisklasse virtuelle Methoden vorweist oder wenn es Methoden mit der gleichen Signatur in mehreren Basisklassen gibt. Moderen Compiler lösen diese Probleme sehr elegant jedoch muss man immer daran denken dass dies nicht ohne zusätzlichen Overhead möglich ist. Auch muss man sich zusätzliche Gedanken machen in welcher Reihenfolge die Konstruktoren der Basisklassen aufgerufen werden man eine Klasse erzeugt welche von mehreren Basisklassen erbt.

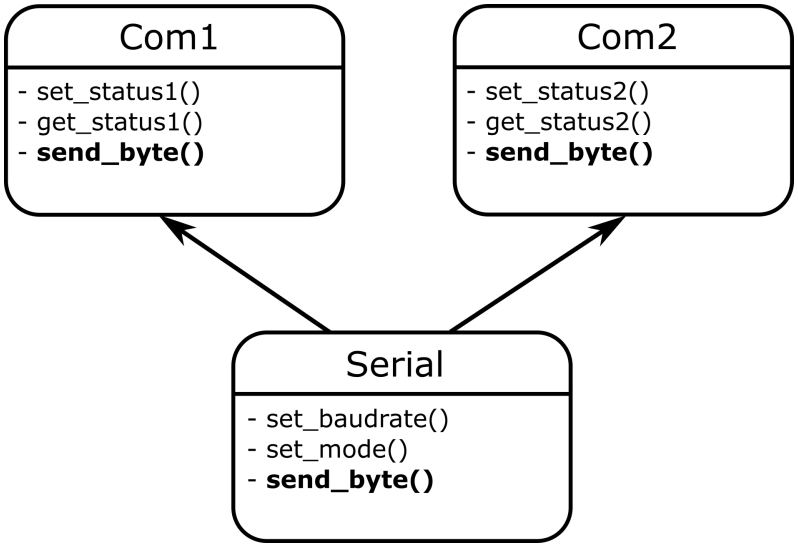

Abbildung 28: Mehrfache Vererbung

Um zu verdeutlichen welche Auswirkungen ein Konstrukt wie in Abbildung 28 dargestellt auf die Größe der Serial Klasse auswirkt analysieren wir die Größe erneut mit *sizeof*. Das Ergebnis von *sizeof(Serial)* ergibt nun 28 Byte was ein Overhead von 8 Byte gegenüber der Version mit nur einer Basisklasse. Die 8 Byte setzen sich aus einer zusätzlichen Variable (4 Byte) welche in der zweiten Basisklasse definiert wurde und einem Zeiger auf eine VFT welche sich nun auch in der *Serial* Klasse befindet. Schlussfolgernd kann man sagen dass mehrfache Vererbung in Kombination mit vielen virtuellen Methoden zu einem hohen Overhead führt und somit in embedded Projekten nicht zur Routine werden soll.

2.8.6 Fazit

Das Verwenden von Klassen und den dazugehörigen Konzepten wie Konstruktoren oder Vererbung kann definitiv ein Mehrwert für ein Projekt darstellen, da man oftmals eine bessere Modularisierung erlangt und die Wartbarkeit erhöht. Auch der Overhead ist oftmals nicht höher als bei einer äquivalenten C Implementierung wenn man auf gewisse Punkte achtet. Bei der Verwendung von virtuellen Methoden und mehrfacher Vererbung sollten bei jedem embedded Entwickler die Alarmglocken läuten und man muss sich Überlegen ob man mit dem Overhead leben kann wenn der Vorteil der dadurch entsteht überwiegt. Jedoch muss man ganz klar sagen dass eine Implementierung in C++ viel mehr Wissen und Know-How von einem embedded Entwickler fordert als eine reine funktionale Implementierung in C.

2.9 Fehlerbehandlung

Fehlerbehandlung mittels *try, catch* ist in embedded Projekten noch sehr exotisch da ein relativ hoher Overhead im Fehlerfall entstehen kann. Jedoch hat diese Art von Fehlerbehandlung einige Vorteile welche hier erläutert werden sollen. Braucht man eine sehr performante Fehlerbehandlung welche sich über mehrere Softwareebenen erstreckt kann die Anwendung der *try, catch* Methode durchaus sinnvoll sein. Somit kann man auch Fehler welche in Konstruktoren auftreten behandeln was aufgrund des fehlenden Rückgabewertes bei Konstruktoren nicht über die übliche Art funktioniert.

Um den Overhead von *try, catch* zu analysieren implementieren wir einmal eine Fehlerbehandlung mittels Rückgabelwert und der *try, catch* Methode um beide Versionen auf Performance und Speicher Overhead zu testen. In der *set_baudrate* Methode soll eine Fehlerüberprüfung implementiert werden um ein falsches setzen der Baudrate zu detektieren.

```
1 class Serial {
2
3   public:
4   void set_baudrate(int baudate){
5     if(baudate <= 0)
6       throw (int)-1;
7
8     this->baudrate = baudrate;
9   }
10
11
12  private:
13  int baudrate;
14 };
```

Quellcode (52) Try, Catch Methode

```
1 class Serial {
2
3   public:
4   int set_baudrate(int baudate){
5     if(baudate <= 0)
6       return -1;
7
8     this->baudrate = baudrate;
9     return 1;
10  }
11
12  private:
13  int baudrate;
14 };
```

Quellcode (53) Return Methode

In der *try, catch* Version ist der Rückgabe Typ der Methode *void* und mittels *throw* kann ein Fehler eines beliebigen Typs geworfen werden welcher dann außerhalb im *try, catch* Block aufgefangen werden muss. In unserer Implementierung wird lediglich ein Fehler vom Typ *int* geworfen. In der Praxis werden häufig eigene Fehler Klassen implementiert welche im Fehlerfall detaillierte Informationen über den Fehler ausgeben oder spezifische Fehler Routinen ausführen. In der klassischen Version mittels Rückgabelwert liefert die Methode (-1) im Fehlerfall und (1) wenn die Baudrate auf einen gültigen Wert gesetzt wurde. In diesem Fall muss das Hauptprogramm den Rückgabelwert der der *set_baudrate* Methode jedes Mal überprüfen um einen möglichen Fehler zu detektieren. Das nachfolgende Beispiel zeigt wie eine Fehlerbehandlung von beiden Versionen im Hauptprogramm ausschauen würde.

```
1 #include "serial.h"
2
3 int main(void){
4
5   Serial ser;
6
7   try{
8     ser.set_baudrate(-1);
9   }
10  catch(int exception){
11    return exception;
12  }
13
14  return 1;
15 }
```

Quellcode (54) Try, Catch Methode

```
1 #include "serial.h"
2
3 int main(void){
4
5   Serial ser;
6
7   if(ser.set_baudrate(-1) == -1){
8     return -1;
9   }
10
11  return 1;
12 }
13
14
15 /*EOF*/
```

Quellcode (55) Return Methode

Die beiden Programme tun exakt das Gleiche unterscheiden sich jedoch in Performance und ROM Größe. Um die Performance zu analysieren wurden die Assembler Instruktion ab Zeile 7 bis zum Programm Ende für den Fehlerfall und den nicht Fehlerfall gezählt. Hierfür wurde das Programm auf einen Mikrocontroller gespielt und im Debug Modus die Instruktionen zwischen 2 Breakpoints gezählt. Mit diesem Test bekommt man ein ungefähres Gefühl wie viel Overhead die jeweilige Fehlerbehandlungs Methode in beiden Fällen hat. Der genaue Performance Overhead hängt natürlich stark von der verwendeten CPU und der Optimierungsstufe des Compilers ab. Tabelle 6 zeigt das Ergebnis dieses Testes.

Methode	**Fehlerfall**	**Nicht Fehlerfall**
try, catch	550	18
return value	25	26

Tabelle 6: Performance Fehlerbehandlung

Wie in der Tabelle 6 ersichtlich ist die *try, catch* Methode sogar um 8 Instruktionen schneller als die herkömmliche Methode mit Rückgabelwert wenn kein Fehler auftritt. Dieser kleine Overhead liegt hauptsächlich an der Überprüfung des Rückgabelwertes welcher bei der return Methode durchgeführt werden muss. Dies ist möglich da moderne C++ Compiler das *zero-cost Model* bei der *try, catch* Methode umsetzen. Dies besagt das im nicht Fehlerfall kein zusätzlicher Code ausgeführt werden soll. Jedoch erkennt man im Fehlerfall warum die *try, catch* Methode in embedded Projekten noch nicht sehr beliebt ist. Im Fehlerfall ergibt sich bei der *try, catch* Methode nämlich ein Overhead von ungefähr 500 Instruktionen was enorm ist. Dieser Overhead entsteht durch eine zusätzliche Exception Tabelle die zur Laufzeit erstellt und gewartet werden muss. Diese Tabelle beinhaltet die Information zu welchem catch Block im Fehlerfall gesprungen werden muss und wird bei jedem Eintritt in ein try Block adaptiert.

2.9.1 Fazit

Bei kleineren Mikrocontrollern macht die *try, catch* Fehlerbehandlung absolut keinen Sinn da der ROM Overhead einfach zu groß ist. Auf Plattformen mit leistungsstärkeren CPU's in Kombination mit genügend ROM Speicher kann die Verwendung einer solchen Fehlerbehandlungs-Methode durchaus eine Überlegung Wert sein. Die geringe Performance Verbesserung bei der *try, catch* Methode kann bei der Berechnung mathematischer Probleme die unter Umständen viel Zeit in Anspruch nehmen kann sogar einen großen zeitlichen Unterschied in der Berechnungszeit bewirken. Schlussfolgernd kann man sagen dass die konventionelle Methode mit Überprüfung des Rückgabelwertes für die meisten Projekte vollkommen ausreichend ist und so schnell nicht aussterben wird.

2.10 Operatoren Überladen

Bei der Verwendung von Klassen welche verschiedene Datentypen beinhalten müssen Operatoren wie *+, +=, »* überschrieben werden damit man diese sinnvoll einsetzen kann. Ist ein Operator in Bezug auf eine bestimmte Klasse sinnvoll sollte man diesen stets überschreiben um die gewünschte Funktionalität zu gewährleisten. Überladene Operatoren führen dazu dass der Code wesentlich übersichtlicher wirkt als müsste man eine eigene Operator Funktion aufrufen. Das nachfolgende Beispiel zeigt wie man den Addier- Operator überschreiben kann und wie eine Implementierung ohne Überschreiben des Operators möglich wäre. Direkt ersichtlich ist das der Aufruf in Zeile 23 wesentlich übersichtlicher wirkt wenn der Operator überladen wurde. Wenn man den *+* Operator überschrieben hat sollte man auch immer den zugehörigen *+=* Operator überschreiben um blöde Fehler im vorhinein zu vermeiden. Implementiert man in C++ eine Klasse welche eine Ausgabe oder Eingabe erfordert sollte man wenn möglich die *»* und *«* Operatoren überschreiben.

```
1  class Class {
2
3    public:
4    int a{0};
5    int b{0};
6
7    Class operator+(const Class& val) {
8      Class tmp;
9      tmp.a = a + val.a;
10     tmp.b = b + val.b;
11     return tmp;
12   }
13 };
14
15 int main(void){
16
17   Class class_a, class_b;
18   Class class_c;
19
20   class_a.a = 2;
21   class_b.b = 5;
22
23   class_c = class_a + class_b;
24
25   return 1;
26 }
```

Quellcode (56) Operator Überladen

```
1  class Class {
2
3    public:
4    int a{0};
5    int b{0};
6
7    Class add(const Class& val) {
8      Class tmp;
9      tmp.a = a + val.a;
10     tmp.b = b + val.b;
11     return tmp;
12   }
13 };
14
15 int main(void){
16
17   Class class_a, class_b;
18   Class class_c;
19
20   class_a.a = 2;
21   class_b.b = 5;
22
23   class_c = class_a.add(class_b);
24
25   return 1;
26 }
```

Quellcode (57) Addier Methode

Das Überladen von Operatoren erzeugt auch keinen Overhead an Performance oder ROM Größe. Der Compiler generiert für beide Versionen exakt das gleiche Ergebnis womit ein Unterschied nur in der Lesbarkeit und Wartbarkeit des Codes besteht.

```
$ arm-none-eabi-g++ -c -Os main.cpp
$ arm-none-eabi-size main.o
...
text    data     bss
16         0       0
...
```

Analyse Operator Überladen

```
$ arm-none-eabi-g++ -c -Os main.c
$ arm-none-eabi-size main.o
...
text    data     bss
16         0       0
...
```

Analyse Addier Mehode

3 Schlussfolgerung

TODO...

Literaturverzeichnis

[1] C von A bis Z (Jürgen Wolf) - ISBN 978-3-8362-1411-7

[2] Grundkurs C++ (Jürgen Wolf) - ISBN 978-3-8362-3895-3

[3] C und C++ für Embedded Systems (Friedrich Bollow, Matthias Homann, Klais-Peter Köhn) ISBN 978-3-8266-1764-5

[4] Effective Modern C++ (Scott Meyers) - ISBN 978-1-4919-0399-5

[5] C++ In der Embedded-Entwicklung (Günter Obiltschnig)

[6] Resource Acquisition Is Initialization (Wikipedia) https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

Abbildungsverzeichnis

Tabellenverzeichnis

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

VTF Virtual Funktion Table