HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Embedded Software Engineering 2
Advanced and Effective C++

Prof. Reto Bonderer
HSR Hochschule für Technik Rapperswil
reto.bonderer@hsr.ch

April 2020

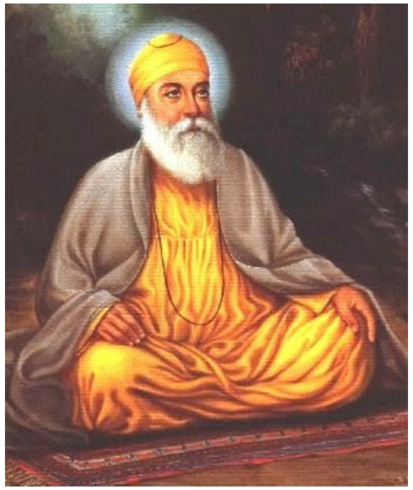
Ziele

- Einführung C++ für Embedded Systems
- Performancebetrachtungen
- Inlining
- pImpl-Idiom

Effective Embedded C++ (vs. C?)

Fallbeispiel aus *Design Patterns for Embedded Systems in C* von "Embedded Guru" Bruce Powell Douglass

```c
/* class Queue */
typedef struct Queue Queue;
struct Queue
{
  int buffer[QUEUE_SIZE];
  int head;
  int tail;
  int size;
  int (*isFull)(Queue* const me);
  int (*isEmpty)(Queue* const me);
  int (*getSize)(Queue* const me);
  void (*insert)(Queue* const me, int k);
  int (*remove)(Queue* const me);
};
```



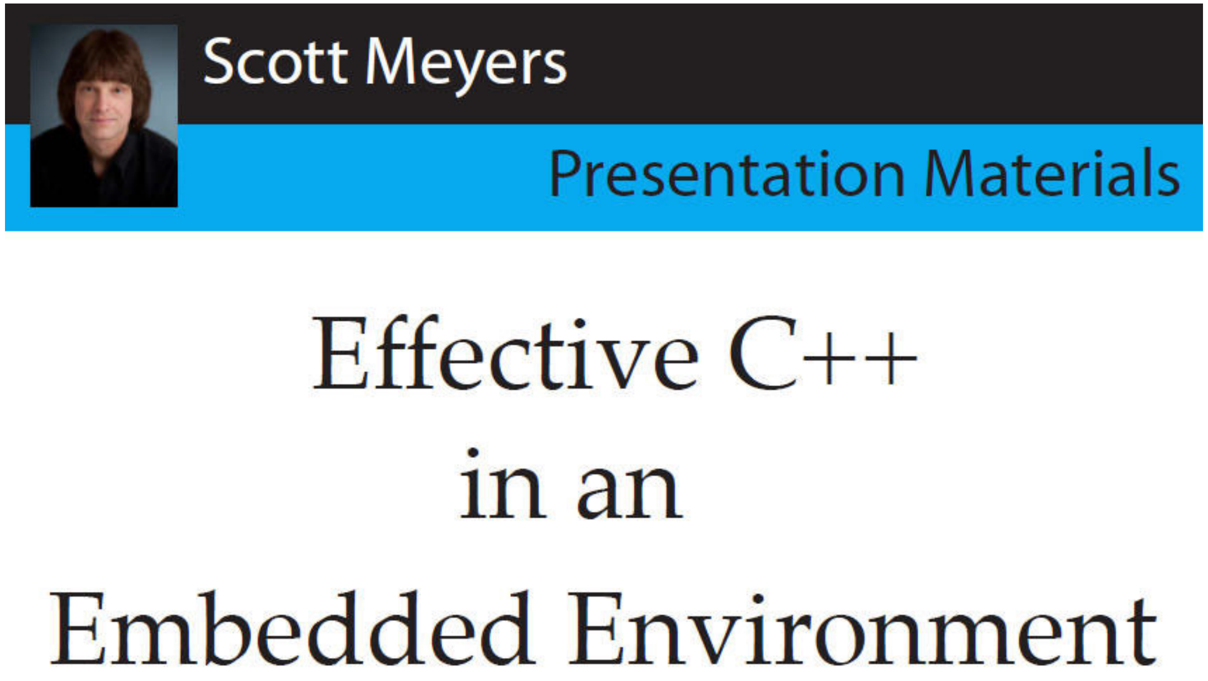
Fallbeispiel aus *Design Patterns for Embedded Systems in C* von "Embedded Guru" Bruce Powell Douglass

```
void Queue_Init(Queue* const me,
                int (*isFullFunction)(Queue* const me),
                int (*isEmptyFunction)(Queue* const me),
                int (*getSizeFunction)(Queue* const me),
                void (*insertFunction)(Queue* const me, int k),
                int (*removeFunction)(Queue* const me));
```

Was ist Ihre Meinung zu dieser Variante?

Effective C++ in an Embedded Environment

Wesentliche Teile der folgenden Informationen stammen aus einem Seminar von Scott Meyers


Scott Meyers

Presentation Materials

Effective C++
in an
Embedded Environment

Embedded Systems

Embedded systems using C++ are diverse:

▪ Real-time?	Maybe.
▪ Safety-critical?	Maybe.
▪ Challenging memory limitations?	Maybe.
▪ Challenging CPU limitations?	Maybe.
▪ No heap?	Maybe.
▪ No OS?	Maybe.
▪ Multiple threads or tasks?	Maybe.
▪ “Old” or “weak” compilers, etc?	Maybe.
▪ No hard drive?	Often.
▪ Difficult to field-upgrade?	Typically.

Developing for Embedded Systems

- In general, little is “special” about developing for embedded systems:
  - Software must respect the constraints of the problem and platform.
  - C++ language features must be applied judiciously.

- These are true for non-embedded applications, too.
  - **Good embedded software development is just good software development.**

Implementing C++

- Why Do You Care?
  - You’re just curious: how do they do that?
  - You’re trying to figure out what’s going on while debugging.
  - You’re concerned: do they do that efficiently enough?
    - That’s the focus of this presentation
    - Baseline: C size/speed

- Have faith:
  - C++ was designed to be competitive in performance with C.
  - Generally speaking, you don't pay for what you don't use.

Performancebetrachtungen (Zeit und Codegrösse)

No-cost C++ features (compared to C)

- All the C stuff: structs, pointers, free functions, etc.
- Classes
- Namespaces
- Static functions and data
- Nonvirtual member functions
- Function and operator overloading
- Default parameters:
  Note that they are *always* passed. Poor design can thus be costly:
  ```
  void doThat(const std::string& name = "Unnamed"); // Bad
  const std::string defaultName = "Unnamed";
  void doThat(const std::string& name = defaultName); // Better
  ```
  Overloading can be a cheaper alternative. (Braucht aber mehr Code)
- Constructors and destructors: They contain code for mandatory initialization and finalization. However, they may yield chains of calls up the hierarchy.
- Single inheritance

Low-cost C++ features (compared to C)

You may pay for these features, even if you don't use them:

- Exceptions: a small speed and/or size penalty (code)
  When evaluating the cost of exceptions, be sure to do a fair comparison. Error handling costs you something, no matter how it is implemented.
  E.g., Saks reports object code increases of 15-40% for error handling based on return values.

C++ Features that can surprise inexperienced C++ programmers

These **can** cost you if you're not careful:

- Temporary objects, e.g., returned from a+b:
    - Many techniques exist to reduce the number and/or cost of such temporaries.

- Templates
    - Gefahr von Code Bloat (mehr dazu später)

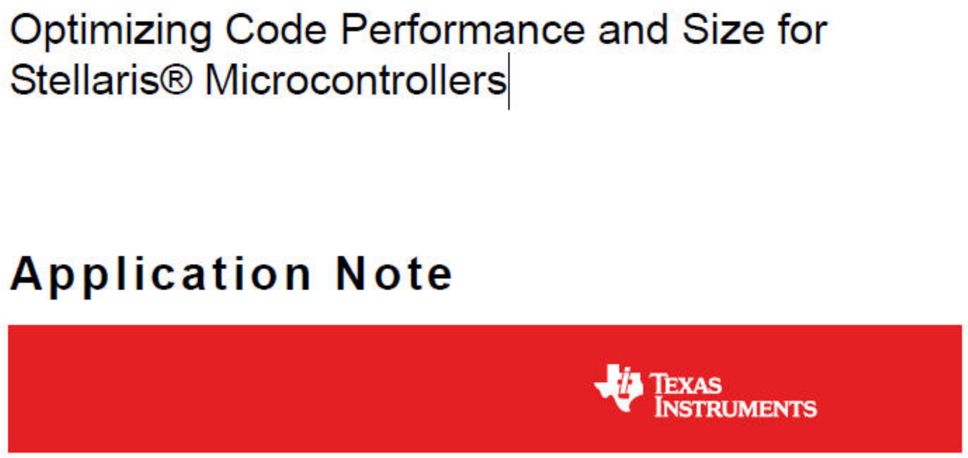
Common Questions

- Why are simple “hello world” programs in C++ so big compared to C?
  - iostream vs. stdio
  - “hello world” is an atypical program:
    - For small programs, C++ programmers can still use stdio
    - This improved a lot in the meantime (compilers and linkers are getting better)
- Why do C developers moving to C++ often find their code is big and slow?
  - C++ isn't C, and C programmers aren't C++ programmers
  - C++ from good C++ developers as good as C from good C developers

Efficiency Beyond C: C++ can be *more efficient* than C

- C++ feature implementation often better than C approximations:
  - E.g., virtual functions
- Abstraction + encapsulation ⇒ flexibility to improve implementations:
  - std::strings often outperform char*- based strings:
    - May use reference counting
    - May employ “the small string optimization”
- STL-proven techniques have revolutionized library design:
  - Shift work from runtime to compile-time:
    - Template metaprogramming (TMP), e.g., “traits”
    - Inlined operator()s
  - Sample success story: C++’s **`sort()`** is faster than C’s **`qsort()`**.

Performance-Betrachtungen



Optimizing Code Performance and Size for Stellaris® Microcontrollers

Application Note

Texas Instruments

Variable Size

Hidden Cost

Using variables larger than the processor is comfortable with means extra loads/stores, extra computation (software routines versus hardware), or far slower instructions. This often ends up as calls, which does not add to size, but does significantly affect performance.

Using variables smaller than optimal may mean extra instructions to sign or unsign extend (on load and after computations). These may also prevent the use of optimal load and store instructions.

Cortex-M3

Long long is generally optimized due to special instructions (for example, ADDC and UMULL). Smaller globals and statics are okay, but locals are best as ints and unsigned ints (or longs). If globals/static are used a lot, copy to int locals for duration, and copy back. It is not unusual to have a 40% increase in function size due to use of short locals.

Variablentypen

- `int` und `unsigned int` sind immer die effizientesten Datentypen
- Sie entsprechen der Registergrösse des jeweils verwendeten Prozessors
- Die Verwendung von `short` oder `char` spart deshalb kaum Speicher, ist jedoch üblicherweise deutlich langsamer als `int`

Variable size: Beispiel

```
typedef int BASE;

BASE foo(BASE last, BASE x, BASE y)
{
 0: 2300               movs r3, #0
 2: e002               b.n a <foo+0xa>
  BASE i;
  for (i = 0; i < last; i++)
    x += (y * x);
 4: fb02 1101          mla r1, r2, r1, r1
 8: 3301               adds r3, #1
 a: 4283               cmp r3, r0
 c: dbfa               blt.n 4 <foo+0x4>
 e: ebc2 0001          rsb r0, r2, r1
  return(x-y);
}
12: 4770               bx lr
```

```
typedef short BASE;

BASE foo(BASE last, BASE x, BASE y)
{
 0: f04f 0c00          mov.w ip, #0; 0x0
 4: e004               b.n 10 <foo+0x10>
  BASE i;
  for (i = 0; i < last; i++)
    x += (y * x);
 6: fb02 1301          mla r3, r2, r1, r1
 a: f10c 0c01          add.w ip, ip, #1 ; 0x1
 e: b219               sxth r1, r3
10: fa0f f38c          sxth.w r3, ip
14: 4283               cmp r3, r0
16: dbf6               blt.n 6 <foo+0x6>
18: ebc2 0001          rsb r0, r2, r1
1c: b200               sxth r0, r0
  return(x-y);
}
1e:                    4770 bx lr
```

Gewährleistung von Portabilität

- Ab **C99** werden im Headerfile `<inttypes.h>`, bzw. `<stdint.h>` verschiedene Typen mit genauer Breite definiert. Es ist deshalb sinnvoll, entweder genau diese Typen zu verwenden oder allenfalls diese mittels `typedef` zu definieren.

- Die Konventionen lauten wie folgt:

  `int`**`N`**`_t`, wobei N = 8, 16, 32 für `signed int`-Typen
  Beispiele: `int8_t`, `int16_t`, `int32_t`

  `uint`**`N`**`_t`, wobei N = 8, 16, 32 für `unsigned int`-Typen
  Beispiele: `uint8_t`, `uint16_t`, `uint32_t`

Ausschnitt aus <stdint.h> von gcc

```c
typedef signed char int8_t;
typedef short int16_t;
typedef long int32_t;
typedef long long int64_t;

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;
```

Weitere Definitionen in <stdint.h>

- Weitere Typdefinitionen beinhalten effiziente Typen mit einer Mindestgrösse und schnelle Typen mit einer Mindestgrösse

```
int_leastN_t
uint_leastN_t

int_fastN_t
uint_fastN_t
```

Beispiel:

`<stdint.h>` von TI

`i++` oder `++i` ?

```
i++;
```

entspricht:

```
{
  tmp = i;
  i += 1;
  "return" tmp;
}
```

```
++i;
```

entspricht:

```
{
  i += 1;
  "return" i;
}
```

Die Postfix-Operatoren bedingen im Prinzip ein tmp-Objekt. Gute Compiler werden das optimieren, schlechte jedoch nicht.

Taking address of local variables

Hidden Cost

Local variables are only allocated to the stack if they have to be. Keeping them in registers (and sharing a register between different ones not used at the same time) gives big gains in performance (that is, not having to do loads and stores). **Taking the address of a local will force it to the stack regardless of optimization level.**

Cortex-M3

All Cortex-M3 compilers support all-in-register locals when optimizations are turned on (size or speed).

Inlining

Inlining: Pros und Cons

- Vorteile:
  - Overhead von Funktionsaufrufen ist weg
  - Schneller
  - Branches verschwinden → pipeline- und cachefreundlich
  - Für sehr kleine Funktionen kann sogar die Codegrösse kleiner werden

- Nachteile
  - Führt üblicherweise zu mehr Code
  - Debugging von inlined Funktionen?

Inlining: Einschränkung

- Compiler dürfen inline ignorieren
- Funktionen, auf die ein Pointer zeigt, werden nicht inlined
- Virtuelle Funktionen werden kaum inlined (inline: compile-time, virtual: run-time)
- "Komplizierte" Funktionen ebenfalls nicht (der jeweilige Compiler entscheidet, was "kompliziert" ist)
- Rekursive Funktionen werden nie inlined
- (Ältere) **Linker** können kaum Inlining, d.h. die inline-Funktionen sollten bereits dem Compiler vollständig bekannt sein. Dies kann erreicht werden, indem die Inline-Funktionen direkt im Header definiert werden

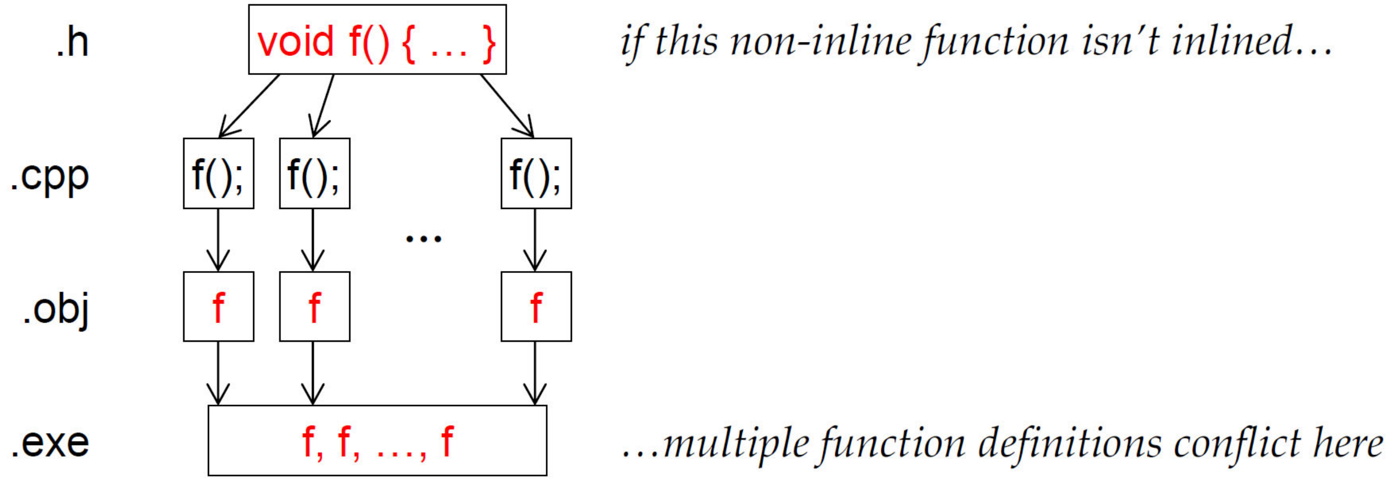
Normal Inlining

- Inline functions are defined in headers
- .cpp files know the function definition by #including the header
    - compiler can inline this function

Automatic Inlining

Compilers may inline functions not declared inline, but this is uncommon.

- To inline a function, compilers need its *definition*, but non-inline functions are not defined in header files.
- They'd cause duplicate symbol errors during linking:



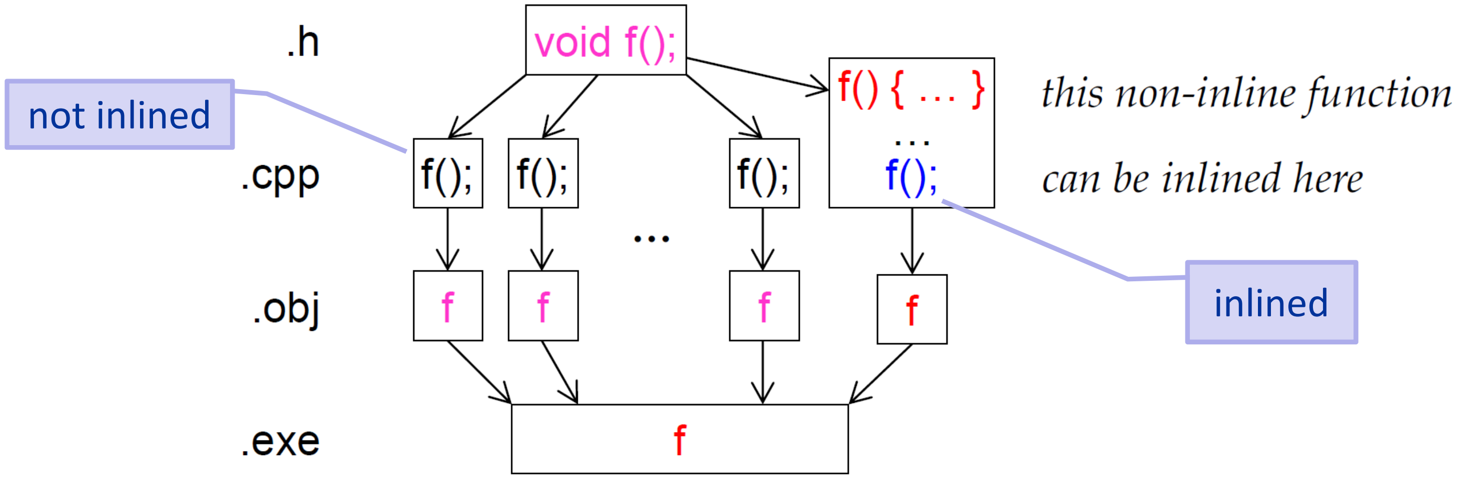
.h
void f() { … }
if this non-inline function isn't inlined…
.cpp
f();
f();
…
f();
.obj
f
f
f
.exe
f, f, …, f
…multiple function definitions conflict here

- **Non-inline functions are thus *declared* in headers, not defined there.**
- The rules for function templates are a bit different….

Automatic Inlining (cont'd)

Compilers rarely inline functions only declared in headers.

- They need to know the function body to inline it.
- When they do know it, inlining is easy (and common).
  - E.g., in the .cpp file defining the function:



Link-Time Inlining

- Linkers are also allowed to perform inlining:
  - Many already do when ***Whole Program Optimization (WPO)***, aka ***Link-Time Optimization (LTO)***, is on, e.g. Gnu, Microsoft, Intel, Sun

- Merely eliminating calls/returns during linking isn't enough.
  - Full benefit requires post-inlining flow analysis.
    - Hence the need for WPO, aka LTO above.

- Still, manual inline declarations remain a necessary evil.

Link-Time Inlining: Bottom Line

- Inlining is almost always a good bet for small, frequently called functions.
  - Overall runtime speed is likely to increase.
- Imprudent inlining can lead to code bloat.
- Minimize inlining if binary upgradeability is important.

pImpl Idiom

`pImpl` Idiom: Problem

- C++ Klassendeklarationen sind durch die Header-Files immer öffentlich, auch private-Teile sind sichtbar
  - unnötige Abhängigkeiten
  - zusätzliche #includes für Membervariablen (wegen Aggregationen) in Header
  - Änderungen am Header (auch in privaten Teilen) bedingen Neu-Compilation aller .cpp Dateien, die den Header einbinden
  - kann Compile-Zeit in grossen Projekten drastisch erhöhen!

- Wie kann man die Implementierung einer Klasse so verstecken, dass man sie ändern kann, ohne alle Module, welche die Klasse nutzen, bei einer Änderung neu übersetzen zu müssen?

`pImpl` Idiom: Lösung

- Man “versteckt” die eigentliche Implementierung der Klasse im .cpp File
- Im Header der Klasse definiert man die öffentliche Schnittstelle und einen privaten Zeiger auf die “versteckte” Implementierung
- `pImpl` - pointer to `Impl`ementation
  - am besten: ein `shared_ptr<Impl>`
- Im Konstruktor der Klasse wird ein `Impl`-Objekt erzeugt
- In der Implementierung der Schnittstelle delegiert man alle Memberfunktionen an die `Impl` Klasse

`pImpl`: Beispiel

```
#ifndef HIDDENCOUNTER_H_
#define HIDDENCOUNTER_H_
#include <memory>

class HiddenCounter
{
public:
  HiddenCounter(int i=0);
  void inc();
  int getCount() const;
  void dec();
  void reset();
private:
  std::shared_ptr<class CounterImpl> pImpl;
};
#endif /* HIDDENCOUNTER_H_ */
```

```
#include "HiddenCounter.h"
class CounterImpl
{
  public:
    CounterImpl(int i): counter(i) {}
    void inc() {++counter;}
    int getCount() const {return counter;}
    void dec() {--counter;}
    void reset() {counter=0; }
  private:
    int counter;
};

HiddenCounter::HiddenCounter(int i)
  :pImpl(new CounterImpl(i)) {}

void HiddenCounter::inc() {pImpl->inc();}

int HiddenCounter::getCount() const
{return pImpl->getCount();}

void HiddenCounter::dec() {pImpl->dec();}

void HiddenCounter::reset() {pImpl->reset();}
```

`pImpl` Idiom in Embedded Systems?

- Die Implementation wird auf dem Heap angelegt
- Das ist unproblematisch, da dieses Objekt beim Programmstart angelegt und erst am Schluss wieder freigegeben wird

- Der `new` Operator könnte bei Bedarf auch überschrieben werden.