HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

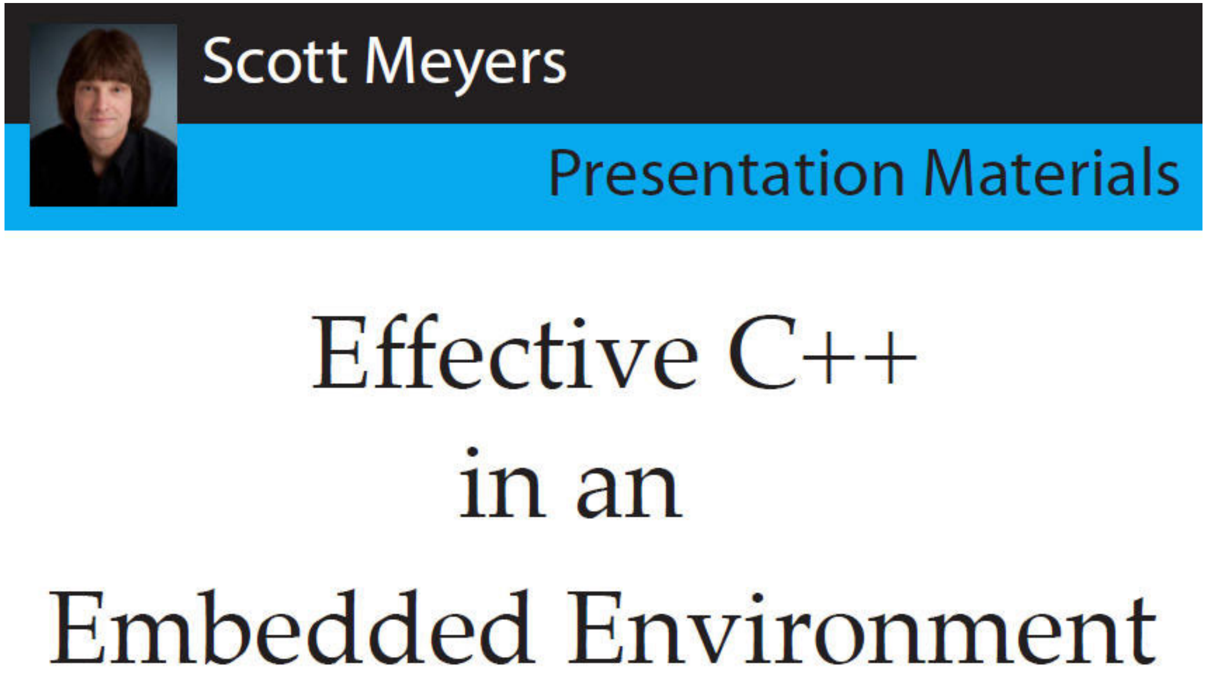
Embedded Software Engineering 2
C++ and ROMability

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

Mai 2020

Effective C++ in an Embedded Environment

Die meisten der folgenden Informationen stammen aus einem Seminar von Scott Meyers



Scott Meyers

Presentation Materials

Effective C++
in an
Embedded Environment

C++ and ROMability

Anything can be burned into ROM and loaded into RAM prior to program execution.

The more interesting question is:

- What may remain in ROM as the program runs?

The C++ Standard is silent on ROMing:

- It allows essentially anything, guarantees nothing.
- What's ROMable is thus up to your compiler and linker.

In what follows, we discuss what is *technically possible*.

- Your compiler/linker probably imposes some restrictions.
  - We'll discuss those first.

C++ and ROM

To understand the restrictions, we need to know what a “POD type” is.

- All C data types are POD (Plain Old Data) types.
- C++11 classes, structs, and unions are generally POD types if they lack:
  - Base classes
  - Virtual functions
  - Non-static data members of reference type
  - User-defined constructors, destructor, or assignment operators
  - Non-static data members of non-POD types

Essentially, a C++11 class or struct is a POD type if it’s “laid out like C and its semantics are preserved if it’s memcpyed.”

- But note that non-virtual member functions are allowed.
- Static data and static member functions are allowed, too.
- The definition of POD types in C++98/03 is stricter, because protected and private nonstatic data members are precluded.

C++ and ROM (cont'd)

Common restrictions on ROMing data:

- Many compilers/linkers will ROM only statically initialized POD types.
  - As we’ll see, it is technically possible for some dynamically initialized non-PODs to be ROMed.
- Some compilers/linkers will ROM structs, but not classes.
  - There is no technical reason for this distinction.
- By the way: ROM is slower than RAM

C++ and ROM (cont'd)

Program instructions can always be ROMed.

Data in a C++ program can be ROMed if it meets two criteria:

- Its value is known before runtime.
  - i.e., either the compiler or the linker knows it or can compute it.

- It can’t be modified at runtime.

C++ and ROM: Examples

```
static const int table[] = {1, 2, 3};     // table is ROMable

const char* pc1 = "Hello World";          // "Hello World" is ROMable
                                          // (but pc1 is not)

const char* const pc2 = "World";          // "World" is ROMable (and may be
                                          // shared with "Hello World");
                                          // pc2 is also ROMable
```

TI: Const types

Hidden Cost

- Global const variables in C may be allocated to SRAM and have their init value in flash.
- In all cases, the value would normally be loaded from memory (unless the compiler can see its initial value).
- Static const scalar variables are like #define macro constants and will not be stored in memory if not needed (address not taken, value small enough to be an immediate).

Cortex-M3

- Different compilers and optimization levels will affect how global const is treated.
- Static const is more reliable for all compilers.
- Enum constants are also a good choice (and can be used with normal ints).

C++ und ROM: int-Konstanten

- Explizit vorhandene int-Konstanten im Sourcecode sollten wegoptimiert werden. Im Normalfall ergibt eine int-Konstante eine Immediate-Adressierung (`MOVE` **`#123,`** `R1`)

- 3 Möglichkeiten für die Definition
  - const int
    - gute Möglichkeit
    - wenn Adresse genommen wird, kann die Konstante nicht wegoptimiert werden
    - u.U. mehrere Kopien im Code
  - **enum**
    - **allerbeste Möglichkeit**
    - **sollte ausschliesslich verwendet werden**
    - (negative Werte sind auch möglich)
  - #define
    - keine Vorteile gegenüber anderen Methoden, vor allem nicht gegenüber enum
    - kein Scope
    - können nicht private oder protected sein

C++ und ROM: double-Konstanten

- double-Konstanten können kaum wegoptimiert werden (Immediate-Adressierung geht normalerweise nicht)
- Ausnahme: bei 64 Bit-Systemen kann es gehen oder bei guter FPU

- 2 Möglichkeiten für die Definition
  - const double
    - gute Möglichkeit
  - #define
    - Makroproblematik
    - private, protected nicht möglich
    - häufig mehrere Kopien dieser Konstanten im Code

C++ and ROM: Objects

- Objects may be ROMed if the following are true:
  - They are declared const at their point of definition.
  - They contain no mutable data members.
  - They are initialized with values known during compilation.
    - Such “knowledge” might come from dataflow analysis, etc.

```
struct Point
{
  int x;
  int y;
};

const Point origin = {0, 0};   // origin is ROMable
```

© HSR Prof. R. Bonderer

C++ and ROM: Compiler Generated Data

Some compiler generated data structures can usually be ROMed:

- Virtual function tables
- RTTI tables and type_info objects
- Tables supporting exception handling

ROMing these objects may be impossible if they are dynamically linked from shared libraries.

Summary: C++ and ROM

- Most compilers/linkers are willing to ROM statically initialized POD types.
  - Aggressive build chains may go beyond this.

- ROMable PODs can be encapsulated by making them protected or private in a non-POD type.

- Compiler-generated data structures are typically ROMable.