HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

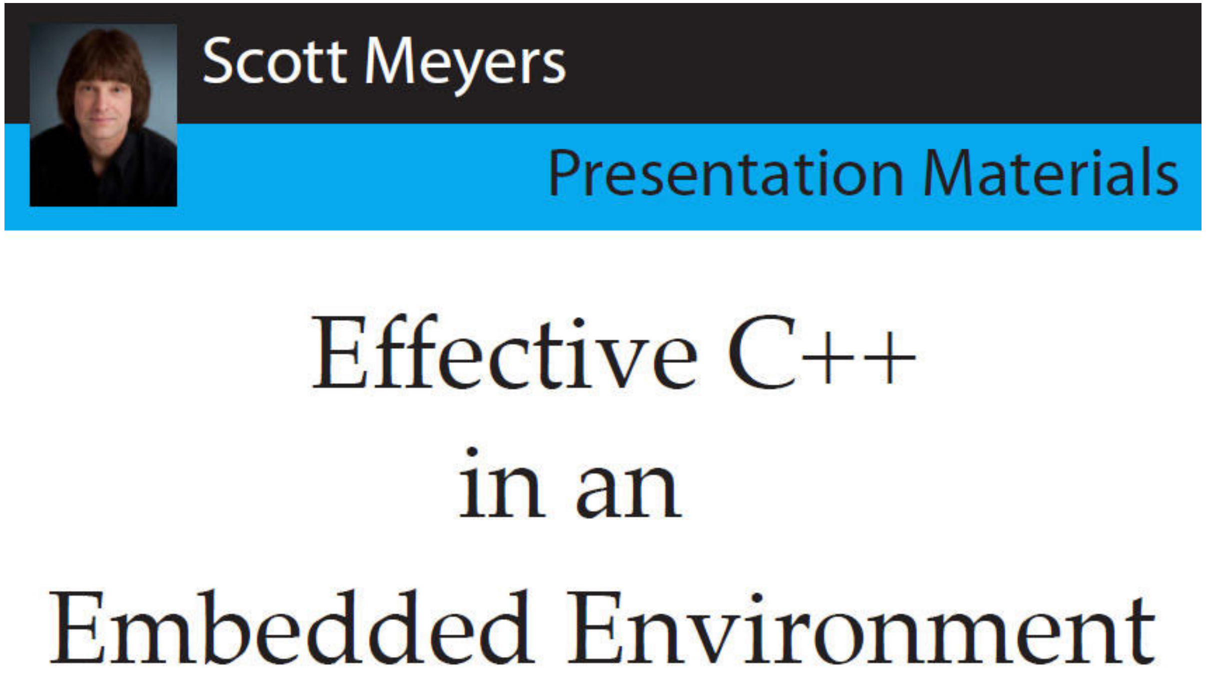
# Embedded Software Engineering 2
# Interface-Based Programming

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

April 2020

## Effective C++ in an Embedded Environment

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



## Interface-Based Programming

*Interface-based programming:*

- **Coding against an interface that allows multiple implementations.**
  - Function interface.
  - Class interface.

- **Client code unaware which implementation it uses.**
  - It depends only on the interface.

# Polymorphism

**The use of multiple implementations through a single interface.**

**Key question: when is it known which implementation should be used?**

- **Runtime**: each call may use a different implementation.
  - Use inheritance + virtual functions.
- **Link-time**: each link may yield a different set of implementations.
  - Use separately compiled function bodies.
  - Applies to both static and dynamic linking.
  - `pImpl` idiom
- **Compile-time**: each compilation may yield a different set of implementations.
  - Use computed typedefs.

## Runtime Polymorphism

- The “normal” meaning of interface-based programming.
  - In much OO literature, the only meaning.
  - Unnecessarily restrictive for C++.
- The most flexible.
  - Can take advantage of information known only at runtime.
- The most expensive.
  - Based on vptrs, vtbls, non-inline function calls.

## Runtime Polymorphism Example

```
class Packet {                          // base class (“interface”)
public:
  …
  virtual bool isWellFormed() const = 0;
  virtual std::string payload() const = 0;
  …
};
class TCPPacket: public Packet {        // derived class (“implementation”)
  …
  bool isWellFormed() const overrride;   // override is C++14
  std::string payload() const override;
  …
};
class CANPacket: public Packet {        // derived class (“implementation”)
  …
  bool isWellFormed() const override;
  std::string payload() const override;
  …
};
```

## Runtime Polymorphism Example (cont'd)

```
std::unique_ptr<Packet> nextPacket( /* params */ );    // factory function; returns next packet
...
std::unique_ptr<Packet> p;
while (p = nextPacket( /* params */), p.get() != nullptr)  // side effect, comma operator
{
  if (p->isWellFormed())                               // use Packet interface
  {
    ...
  }
  ...
}
```

**Runtime polymorphism is reasonable here:**

- Types of packets vary at runtime.

## Link-Time Polymorphism

- Useful when information already known during linking, but not yet during compilation.
- No need for virtual functions.
- Typically disallows inlining.
  - Most inlining is done during compilation.

## Link-Time Polymorphism Example

Software can be deployed on two kinds of boxes:

- Expensive, high-performance box.
  - Uses expensive, fast components.
- Cheaper, lower-performance box.
  - Uses cheaper, lower-performance components.
- Essentially the same software runs on both boxes.
- Component driver implementations differ.
  - A common interface can be defined.

Approach:

- One class definition for both drivers.
- Different component-dependent implementations.
- Implementations selected during linking.
  - This is “C” polymorphism.

# Link-Time Polymorphism Example

**device.h:**

```
namespace Drivers
{
class Impl;                 // forward declaration
class DeviceDriver          // all nonvirtual non-inline functions
{
  public:
    DeviceDriver();
    ~DeviceDriver();
    void reset();
    ...
  private:
    Impl* pImpl;            // ptr to data for driver
};
}
```

All client code `#include`s this header and codes against this class.

- Note lack of virtual functions.

# Link-Time Polymorphism Example (cont'd)

`EFDevice.cpp` (generates `EFDevice.o`, `EFDevice.obj`, or `EFDevice.dll`, etc.):

- EFDevice = “Expensive Fast Device”

```
namespace Drivers
{
struct Impl { ... };               // data needed by EFDevice driver
DeviceDriver::DeviceDriver()       // ctor code for EFDevice
{ ... }
DeviceDriver::~DeviceDriver()      // dtor code for EFDevice
{ ... }
void DeviceDriver::reset()         // reset code for EFDevice
{ ... }
...
}
```

All functions in this file have access to the `Impl` struct defined here.

# Link-Time Polymorphism Example (cont'd)

**<u>CSDevice.cpp (generates CSDevice.o, CSDevice.obj, or CSDevice.dll, etc.):</u>**

CSDevice = “Cheap Slow Device”

```
namespace Drivers
{
struct Impl { ... };                // data needed by CSDevice driver
DeviceDriver::DeviceDriver()        // ctor code for CSDevice
{ ... }
DeviceDriver::~DeviceDriver()       // dtor code for CSDevice
{ ... }
void DeviceDriver::reset()          // reset code for CSDevice
{ ... }
...
}
```

All functions in this file have access to the `Impl` struct defined here.

- `Impl` in this file typically different from that in `EFDevice.cpp`.
- Function bodies in this file also typically different.

## Link-Time Polymorphism Example (cont'd)

Link with:

- `EFDevice.o` if building for expensive, high-performance box.
  - Or link dynamically with e.g. `EFDevice.dll`.
- `CSDevice.o` if building for cheaper, lower-performance box.
  - Or link dynamically with e.g. `CSDevice.dll`.

Link-time polymorphism is reasonable here:

- Deployment platform unknown at compilation, known during linking.
  - No need for flexibility or expense of runtime polymorphism.
    - No vtbls.
    - No indirection through vtbls.
    - No inheritance needed.

## Compile-Time Polymorphism

- Useful when
  - Implementation determinable during compilation.
  - Want to write mostly implementation-independent code.

- No need for virtual functions.
- Allows inlining.
- Based on *implicit interfaces*
  - Other forms of polymorphism based on *explicit interfaces*.

## Device Example Reconsidered

Goal:

- Device class to use determined by platform's #bits/pointer, e.g. 16 vs. 32 bits.
  - This is known during compilation.

Approach:

- Create 2 or more classes with "compatible" interfaces.
  - i.e., support the same implicit interface.
  - e.g., must offer a reset function callable with 0 arguments.
- Use compile-time information to determine which class to use.
- Define a `typedef` for this class.
- Program in terms of the `typedef`.

## Compile-Time Polymorphism Example

**Revised device.h:**

```
#include "NASDevice.h"      // NAS = "Normal Address Space" (32 bits);
                            // defines class NASDevice
#include "BASDevice.h"      // BAS = "Big Address Space" (>32 bits);
                            // defines class BASDevice
#include "SASDevice.h"      // SAS = "Small Address Space" (<32 bits);
                            // defines class SASDevice
...                         // remainder of device.h (coming soon)
```

By design, each class has a compatible interface.

- Members with identical names, compatible types, etc.

## Compile-Time Polymorphism Example (cont'd)

Driver classes may use any language features:

- **Especially inlining.**

```
class NASDevice {
public:
   ...
   void reset() { ... }          // inline function
   ...
};
class BASDevice {
public:
   ...
   void reset() { ... }          // inline function
   ...
};
class SASDevice {
   ...
   void reset();                 // non-inline function
   ...
};
```

## Compile-Time Polymorphism Example (cont'd)

Clients refer to the correct driver type this way:

```
Device::type d;              // d's type is either NASDevice, BASDevice, or SASDevice,
d.reset();                   // depending on # of bits/pointer
```

- `Device` "computes" the proper class for type to refer to.
  - Implementation on next page.

Compile-time polymorphism is reasonable here:

- Device type can be determined during compilation.
  - No need for flexibility or expense of runtime polymorphism.
  - No need to configure linker behavior or give up inlining.

# Compile-Time Polymorphism Example (cont'd)

**Revised device.h (cont'd):**

```
template<int ptrBitsVs32> struct DeviceChoice;
template<> struct DeviceChoice<-1> {        // When bits/ptr < 32
  typedef SASDevice type;
};
template<> struct DeviceChoice<0> {         // When bits/ptr == 32
  typedef NASDevice type;
};
template<> struct DeviceChoice<1> {         // When bits/ptr > 32
  typedef BASDevice type;
};
struct Device {
  enum { bitsPerVoidPtr = CHAR_BIT * sizeof(void*) };
  enum { ptrBitsVs32 = bitsPerVoidPtr > 32 ?  1 :
                       bitsPerVoidPtr == 32 ? 0 :
                                             -1    };
  typedef DeviceChoice<ptrBitsVs32>::type type;
};
```

## Summary: Interface-Based Programming

- One interface, multiple implementations.

- Polymorphism used to select the implementation.
  - Runtime polymorphism uses virtual functions.
  - Link-time polymorphism uses linker configuration.
  - Compile-time polymorphism uses computed typedefs