# Effektives C++ in Embedded Systems

**Scott Meyers, Ph.D.**
Fachberater für Softwareentwicklung

smeyers@aristeia.com
http://www.aristeia.com/

Tel: +1 503-638-6028
Fax: +1 503-974-1887


© 2011 Scott Meyers, alle Rechte vorbehalten.


## Wichtig!

In diesem Vortrag nehme ich an, dass Sie alles über C++ wissen.

Möglicherweise stimmt das nicht.

**Wenn Sie etwas Unbekanntes sehen oder hören,**
***Fragen Sie bitte nach*!**


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Überblick

Tag 1 (voraussichtlich):

- "Embedded Systems"
- Ein tiefer Blick in C++
  - ➧ Implementierung von Features der Sprache
  - ➧ Inline-Funktionen
  - ➧ Vermeiden von "Code Bloat"
- 3 Ansätze für Interface-basierte Programmierung
- Dynamische Speicherverwaltung
- C++ und ROM (falls die Zeit reicht)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Überblick

Tag 2 (voraussichtlich):

- Modellierung von memory-mapped I/O
- Implementierung von Callbacks für C Interfaces
- Interessante Anwendungen von Templates:
  - ➧ Typsichere `void*`-basierte Container
  - ➧ Analyse von physikalischen Maßeinheiten zur Kompilierzeit
  - ➧ Die Spezifikation von endlichen Automaten (falls die Zeit reicht)
- Überlegungen für sicherheitskritische und Echtzeitsysteme
- Weiterführende Informationen (Englisch)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Immer wichtig

- **Ihre Fragen, Kommentare, Themen, usw.**
  - ➧ Immer höchste Priorität.

Ziel des Kurses ist, die Themen anzusprechen, die Sie interessieren.

- Egal ob sie in den Folien sind oder nicht.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Überblick

Tag 1 (voraussichtlich):

- "Embedded Systems"
- Ein tiefer Blick in C++
  - ➧ Implementierung von Features der Sprache
  - ➧ Inline-Funktionen
  - ➧ Vermeiden von "Code Bloat"
- 3 Ansätze für Interface-basierte Programmierung
- Dynamische Speicherverwaltung
- C++ und ROM (falls die Zeit reicht)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## "Embedded Systems"

Embedded Systems sind unterschiedlich:

| | |
|---|---|
| ■ Echtzeit? | Vielleicht. |
| ■ Sicherheitskritisch? | Vielleicht. |
| ■ Anspruchsvolle Speicherbeschränkungen? | Vielleicht. |
| ■ Anspruchsvolle CPU Beschränkungen? | Vielleicht. |
| ■ Kein Heap? | Vielleicht. |
| ■ Kein Betriebssystem? | Vielleicht. |
| ■ Mehrfache Threads oder Tasks? | Vielleicht. |
| ■ "Alte" oder "schwache" Compiler? | Vielleicht. |
| ■ Keine Festplatte? | Oft. |
| ■ Schwierig nach der Auslieferung zu aktualisieren? | Normalerweise. |


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Entwicklung für Embedded Systems

Entwicklung für Embedded Systems ist selten "besonders:"

- Software muss das Problem und die Beschränkungen der Plattform berücksichtigen.
- C++ Features müssen sorgfältig verwendet werden.

Das gilt auch für nicht-embedded Anwendungen.

- **Gute Softwareentwicklung für Embedded Systems ist einfach gute Softwareentwicklung.**


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Überblick

Tag 1 (voraussichtlich):

- "Embedded Systems"
- Ein tiefer Blick in C++
  - ➧ Implementierung von Features der Sprache
  - ➧ Inline-Funktionen
  - ➧ Vermeiden von "Code Bloat"
- 3 Ansätze für Interface-basierte Programmierung
- Dynamische Speicherverwaltung
- C++ und ROM (falls die Zeit reicht)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die Implementierung von C++

Warum ist das interessant?

- Neugier: Wie sind die Features der Sprache umgesetzt?
- Sie möchten besser verstehen, was Sie beim Debugging sehen.
- Bedenken: ist die Umsetzung effizient genug?
  - ➧ Der Schwerpunkt dieser Präsentation.
  - ➧ Referenz: Größe und Geschwindigkeit von C.

Haben Sie Vertrauen:

- C++ wurde entworfen, beinah so effizient wie C zu sein.
- Normalerweise bezahlt man nichts für Features, die man nicht nutzt.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Die Implementierung von virtuellen Funktionen

Implementierungen sind etwas unterschiedlich:

- Compiler dürfen virtuelle Funktionen umsetzen, wie sie wollen.
  - ➧ Es gibt keine "einzig richtige" Implementierung.
- Meine Beschreibung ist *meistens* wahr für die meisten Umsetzungen.
  - ➧ Ich ignoriere einige Details.
  - ➧ Nichts davon widerspricht der Tatsache, dass virtuelle Funktionen *sehr* effizient implementiert sind.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die Implementierung von virtuellen Funktionen

Sehen wir diese Klasse an:

```
class B {
public:
  B();

  virtual ~B();
  virtual void f1();
  virtual int f2(char c) const;
  virtual void f3(int x) = 0;

  void f4() const;
  ...
};
```

Compiler nummerieren die virtuellen Funktionen in der Reihenfolge, in der sie deklariert sind. In diesem Beispiel,

- der Destruktor ist Nummer 0.
- f1 ist Nummer 1, f2 ist Nummer 2, f3 ist Nummer 3.

Nicht-virtuelle Funktionen bekommen keine Nummer.

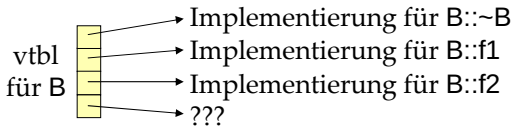

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Die Implementierung von virtuellen Funktionen

Eine *vtbl* ("virtual table") wird für die Klasse erzeugt. Sie sieht etwa so aus:




Bemerkungen:

- Die vtbl ist ein Array, das Funktionszeiger enthält.
- Diese zeigen auf die Implementierungen von virtuellen Funktionen.
  - ➡ Die *n*-te Stelle zeigt auf die virtuelle Funktion mit der Nummer *n*.
  - ➡ Der Inhalt der Stelle für eine rein virtuelle Funktion ist undefiniert.
    - ◆ Ziel ist oft eine Funktion, die einen Fehler meldet und abbricht.
- Nicht-virtuelle Funktionen (darunter Konstruktoren) sind ausgelassen:
  - ➡ Nicht-virtuelle Funktionen sind umgesetzt wie in C.

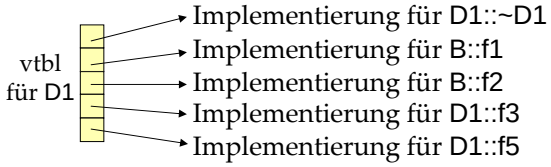

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die Implementierung von virtuellen Funktionen

Jetzt sehen wir uns eine abgeleitete Klasse an:

```
class D1: public B {
public:
  D1();                                  // nicht-virtuell
  virtual void f3(int x);                // überschreibt virtuelle in Basisklasse
  virtual void f5(const std::string& s); // neu virtuell
  virtual ~D1();                         // überschreibt virtuelle in Basisklasse
  ...
};
```

Eine vtbl wie diese wird erzeugt:





Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.

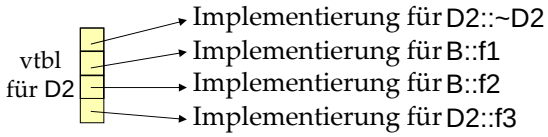


© 2011 Scott Meyers, alle Rechte vorbehalten.

## Die Implementierung von virtuellen Funktionen

Eine zweite abgeleitete Klasse würde ähnlich behandelt:

```
class D2: public B {
public:
  D2();
  virtual void f3(int x);
  ...
};
```

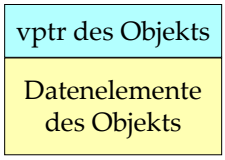



- D2s Destruktor wird vom Compiler automatisch erzeugt.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die Implementierung von virtuellen Funktionen

Objekte von Klassen mit virtuellen Funktion enthalten einen Zeiger auf die vtbl der Klasse:




Dieser Zeiger wird *vptr* ("virtual table pointer") benannt.

- Seine Position im Objekt darf je nach Compiler unterschiedlich sein.

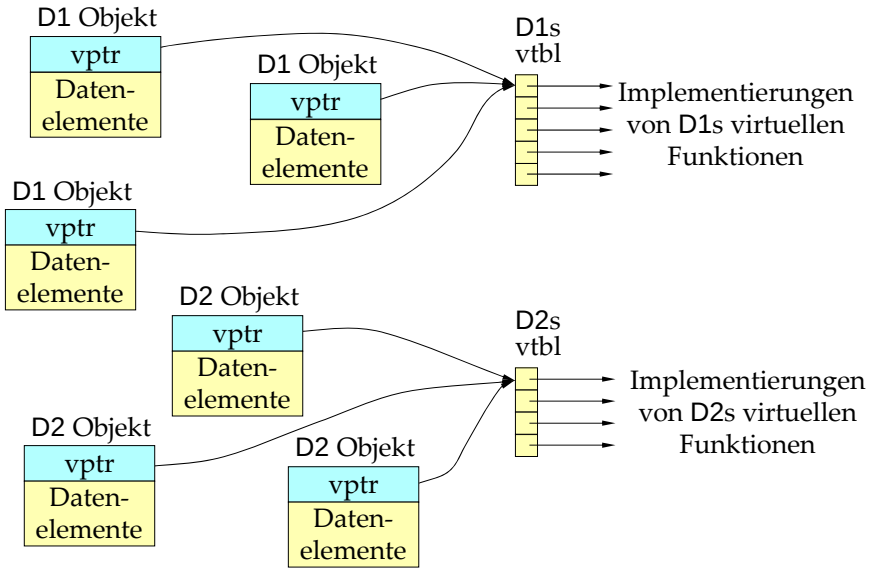

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Die Implementierung von virtuellen Funktionen

Vptrs zeigen auf vtbls:





Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die Implementierung von virtuellen Funktionen

Für folgendes Beispiel

```
void makeACall(B *pB)
{
  pB->f1();
}
```

erzeugt der Compiler so etwa diese Umsetzung:

```
(*pB->vptr[1])(pB);          // rufe die Funktion auf,
                             // die von Stelle 1 gezeigt wird in der vtbl,
                             // die von pB->vptr gezeigt wird;
                             // pB wird als “this” Zeiger übergeben
```

Eine Auswirkung:

- Wenn eine virtuelle Funktion geändert wird, müssen alle Aufrufer neu kompiliert werden!
  - ➡ z.B. wenn die Stelle der Funktion in der Reihenfolge der Klasse geändert wird.
    - ◆ d.h. ihre vom Compiler vergebene Nummer.
  - ➡ z.B. wenn die Signatur der Funktion geändert wird.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.

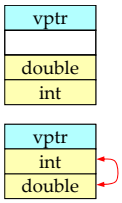


© 2011 Scott Meyers, alle Rechte vorbehalten.

## Kosten der Implementierung

Erhöhung des Speicherbedarfs:

- Vptr lässt Objekte größer werden.
  - ➧ Ausrichtungsbeschränkungen können zu Padding führen.
    - ◆ Oft löst eine Änderung der Reihenfolge der Datenelemente das Problem.




- Vtbls vergrössern Speicherbedarf pro Anwendung.

Geschwindigkeitsnachteile:

- Aufruf durch vtbl langsamer als direkter Aufruf:
  - ➧ Normalerweise nur wenige zusätzliche Maschinenbefehle.
- Inlining im Allgemeinen unmöglich:
  - ➧ Oft sowieso unvermeidbar für virtuelle Funktionen.

Im Vergleich mit Alternativen in C:

- *schneller und kleiner* als if/then/else oder switch-basierte Ansätze.
- garantiert, immer richtig implementiert zu sein.

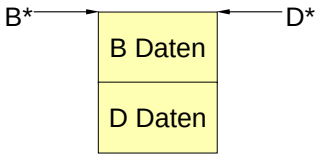

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Objektadressen und Mehrfachvererbung

Unter EV können wir uns das Layout eines Objekts so vorstellen:

```
class B { ... };

class D: public B { ... };
```



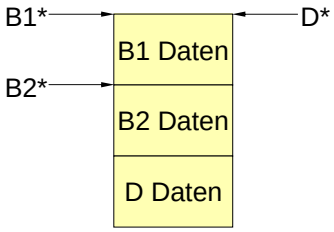

- Eine Ausnahme (bei einigen Compilern) besteht, wenn D virtuelle Funktionen hat, aber B nicht.

Unter MV sieht es eher so aus:

```
class B1 { ... };

class B2 { ... };

class D: public B1,
         public B2 { ... };
```




- D Objekte haben mehrere Adressen:
  - ➧ eine für B1* und D* Zeiger.
  - ➧ eine andere für B2* Zeiger.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.

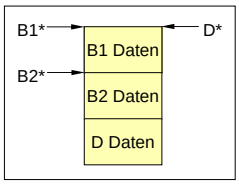


© 2011 Scott Meyers, alle Rechte vorbehalten.

# Objektadressen und Mehrfachvererbung

Mit gutem Grund:

```
void f(B1 *pb1);        // f erwartet, dass pb1 auf ein
                        // B1 Objekt zeigt
void g(B2 *pb2);        // g erwartet, dass pb2 auf ein
                        // B2 Objekt zeigt
```




Deswegen benötigen einige Aufrufe *Adressänderungen*:

```
D *pd = new D;          // keine Adressänderung nötig
f(pd);                  // keine Adressänderung nötig
g(pd);                  // D* ⇒ B2* Änderung nötig
B2 *pb2 = pd;           // D* ⇒ B2* Änderung nötig
```

Korrekte Änderungen erfordern richtige Typinformationen:

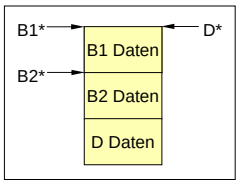
```
if (pb2 == pd) …                // erfolgreich (pd wird zu B2* konvertiert)
if ((void*)pb2 == (void*)pd) …  // pd und pb2 sind nicht gleich
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# Virtuelle Funktionen und Mehrfachvererbung

Eine Herausforderung für Compiler:

```
class B1 {
public:
  virtual void mf();    // möglicherweise überschrieben
  ...                   // in abgeleiteten Klassen
};
class B2 {
public:
  virtual void mf();    // möglicherweise überschrieben
  ...                   // in abgeleiteten Klassen
};
void g(B2 *pb2)         // wie vorher
{
  pb2->mf();            // Adressänderung nötig vor dem Aufruf von mf?
}
```




Änderung ist nur nötig, wenn D mf überschreibt *und* pb2 auf ein D zeigt.

Das Problem des Compilers: Wenn der Code für den Aufruf erzeugt wird,

- weiss er eventuell nicht, ob D existiert oder nicht.
- kann er nicht wissen, ob pb2 auf ein D zeigt oder nicht.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



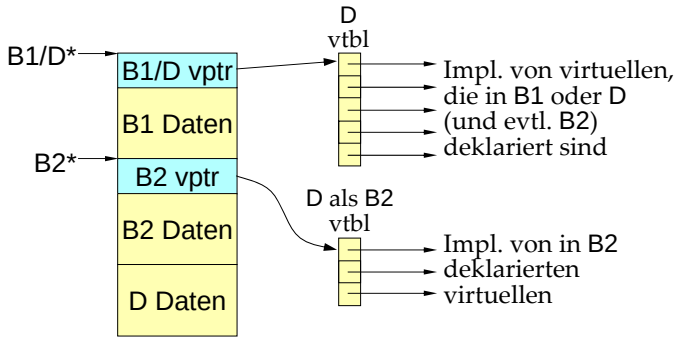
© 2011 Scott Meyers, alle Rechte vorbehalten.

## Virtuelle Funktionen und Mehrfachvererbung

Typische Lösung:

- spezielle vtbls, die Adressänderungen erledigen.
- neue vptrs in Objekten abgeleiteter Klassen, ein zusätzlicher vptr pro Basisklasse nach der ersten:

```
class B1 { … };
class B2 { … };
class D:
   public B1,
   public B2 { … };
```




Diese speziellen vptrs und vtbls gelten nur für Objekte abgeleiteter Klassen.

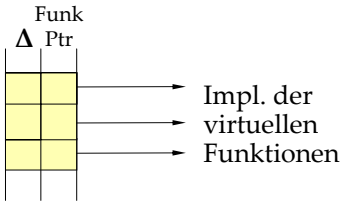
- Virtuelle Funktionen für B1 und B2 Objekte werden wie zuvor implementiert.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Virtuelle Funktionen und Mehrfachvererbung

Adressänderungen werden unterschiedlich umgesetzt:

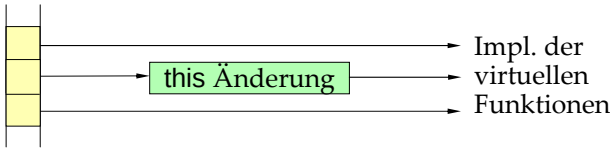
- Das Speichern von Deltas in der vtbl:




```
(*pObjekt->vptr[Index])(pObjekt+Δ);
```

➡ Normalerweise sind die meisten Deltas 0, besonders unter EV.

- Das Weiterleiten von virtuellen Aufrufen durch Thunks:




➡ Thunks werden nur erzeugt, wenn sie benötigt werden.
➡ Dieser Ansatz ist gebräuchlicher.

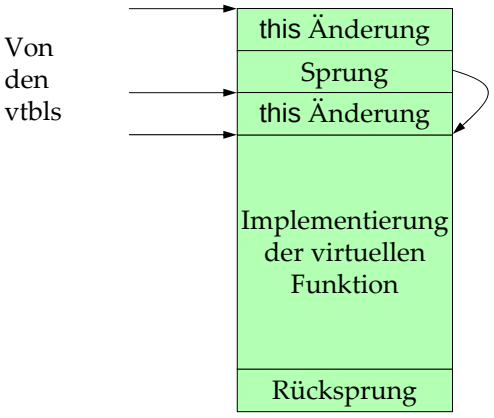

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Implementierung von Thunks

Typischerweise eine Funktion mit mehreren Einstiegspunkten:





Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Virtuelle Funktionen, MV und virtuelle Basisklassen

Der allgemeinste Fall umfasst:

- virtuelle Basisklassen mit nicht statischen Datenelementen.
- virtuelle Basisklassen, die von anderen virtuellen Basisklassen vererben.
- eine Mischung von virtueller und nicht virtueller Vererbung in einer einzigen Hierarchie.

Lippman gibt auf:

> *Die Unterstützung der virtuellen Basisklassen wandert ins Byzantinische...*
> *Das Thema ist einfach zu esoterisch, um diskussionswürdig zu sein...*

Ich gebe auch auf :-)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## "Kostenlose" C++ Features

Tragen nur Kosten zur Kompilierzeit. Im Objektcode sind sie wie C:

- alles von C: structs, Zeiger, freie Funktionen, usw.
- Klassen
- Namespaces
- statische Funktionen und Daten
- nicht-virtuelle Member-Funktionen
- Überladen von Funktionen und Operatoren
- Standardparameter:
  - ➡ Sie werden *immer* übergeben. Guter Entwurf ist deshalb wichtig:

```
void doThat(const std::string& name = "Unnamed");          // schlecht

const std::string defaultName = "Unnamed";
void doThat(const std::string& name = defaultName);        // besser
```

  - ➡ Überladung ist normalerweise eine billigere Alternative.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Mehr "kostenlose" C++ Features

Scheinen Kosten zu tragen, aber eigentlich haben sie keine (im Vergleich mit entsprechendem Verhalten in C):

- Konstruktoren und Destruktoren:
  - ➡ Sie enthalten Code für *zwangsläufige* Initialisierung/Finalisierung.
  - ➡ Sie können aber Aufrufsketten in der Hierarchie verursachen.
- einfache Vererbung
- virtuelle Funktionen
  - ➡ Abstrakte Klassen ohne Implementierungen von virtuellen Funktionen (d.h. "Interfaces") dürfen vtbls noch erzeugen.
    - ◆ Einige Compiler bieten Möglichkeiten an, das zu verhindern.
  - ➡ Nie aufgerufene virtuelle Funktion sind noch eingebunden.
- virtuelle Vererbung


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Noch mehr "kostenlose" C++ Features

- new und delete:
  - Standardmäßig:
    new = malloc + Konstruktor(en) und
    delete = Destruktor(en) + free
  - Fehler-Behandlung durch Exceptions erfolgt automatisch.

Achtung: new ist sogar in Systemen nützlich, in denen alle Speicher statisch alloziert sind.

- *Placement* new ermöglicht, dass Objekte an bestimmten Adressen geschaffen werden können:
  - z.B. im statisch allozierten Speicher.
  - z.B. an memory mapped Adressen.
- später sehen wir Beispiele.

Die bisher genannten Features kosten nur etwas, wenn sie benutzt werden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## C++ Features mit geringen Kosten

Sie fordern eventuell Ressourcen, auch wenn sie nicht benutzt werden:

- Exceptions: ein kleiner Aufwand zur Größe oder Geschwindigkeit des Codes.
  - Wenn Sie die Kosten von Exceptions berücksichtigen, stellen Sie sicher, dass der Vergleich fair ist.
  - Fehlerbehandlung kostet etwas, egal wie sie implementiert wird.
    - z.B. laut Saks steigt die Code-Größe 15-40% für Fehlerbehandlung durch Rückgabewerte.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Ansätze zur Ausnahmebehandlung

Denken Sie über das Problem nach, lokale Objekte abzuräumen:

```
{
  V1
  ...
  {
    ...
    V2
    V3
    ...
  }
  ...
  V4
  V5
  ...
}
```

Welche Objekte müssen abgeräumt werden, falls eine Exception auftritt?

- Es gibt zwei grundsätzliche Ansätze, dieses Problem zu lösen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Ansätze zur Ausnahmebehandlung

Einer davon ist, einen Shadow-Stack der Objekte herzustellen, die im Fall einer Exception abgeräumt werden müssen:

- Größe des Codes nimmt zu, um die Befehle zum Handhaben des Shadow-Stacks zu speichern.
- Bedarf an Laufzeitspeicher steigt durch den Shadow-Stack.
- Laufzeit nimmt zu, um den Shadow-Stack zu manipulieren.
- Leistungsauswirkung?
  - ➧ Unbekannt. Gültige Vergleiche sind schwer zu finden.
  - ➧ Grobe Schätzung: 5-10% mehr Zeit und Speicher.

Dieser Ansatz wird manchmal als “Code Approach” bezeichnet.

- Microsoft verwendet ihn für 32-bit (aber nicht 64-bit) Code.
- g++ auf Windows benutzt ihn auch.

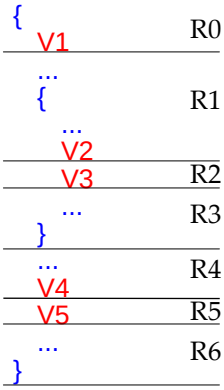

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Ansätze zur Ausnahmebehandlung

Alternativ werden Programregionen gebildet, die Sätzen von Objekten entsprechen, die abgeräumt werden müssen:




| Region | Objekte |
|---|---|
| R0 | Keine |
| R1 | V1 |
| R2 | V1, V2 |
| R3 | V1, V2, V3 |
| R4 | Gleich wie R1 |
| R5 | V1, V4 |
| R6 | V1, V4, V5 |

- Diese Analyse ignoriert, dass Destruktoren Exceptions werfen dürfen.
- Die meisten Compiler unter Unix verwenden diesen Ansatz.
  - ➡ Die 64-bit Itanium ABI benutzt ihn auch.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Ansätze zur Ausnahmebehandlung

Auswirkungen vom "Table Approach":

- Programmgeschwindigkeit unbeeinflusst im Fall keiner Exceptions.
- Programmgröße steigt, da der Code, der die Tabellen handhabt, gespeichert werden muss.
- Statische Programmgröße wächst, um die Tabellen zu speichern.
  - ➡ Falls keine Exception auftritt, müssen die Tabellen nicht im Speicher, im Working-Set oder Cache sein.
- Exceptions zu werfen ist *langsam*:
  - ➡ Tabellen müssen gelesen werden, nachdem sie möglicherweise eingelagert und/oder dekomprimiert werden.
  - ➡ Aber Exceptions sollten nur selten auftreten.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Exceptions und dynamisch allokierter Speicher

Einige Compiler versuchen, Heap-Speicher für Exception-Objekte zu verwenden.

- Das kann in einigen Embedded Systems inakzeptabel sein.

Keine Angst:

- Implementierungen halten nicht-Heap Speicher für Exception-Objekte zurück.
  - ➡ Sie müssen std::bad_alloc Exceptions übertragen können.
  - ➡ Plattformen ohne Heap sollten Exceptions benutzen können.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## C++ Features mit geringen Kosten

Weitere Features, die Kosten haben, auch wenn sie nicht genutzt werden:

- Mehrfachvererbung: kleiner Speicheraufwand (vtbls, die Δs speichern)
- dynamic_cast und andere RTTI Features: kleiner Speicheraufwand (vtbls)
  - ➡ dynamic_cast trägt Kosten, die linear zu der Zahl der Basisklassen (direkt und indirekt) sind, die das gecastete Objekt hat.
    - ◆ Jeder Cast könnte strcmp für jede Klasse der Hierarchie aufrufen.
  - ➡ Implementierungen sind unterschiedlich.
    - ◆ Details finden Sie im *Technical Report on C++ Performance*.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C++ Features, die überraschen können

Das kann teuer werden, wenn Sie nicht aufpassen:

- temporäre Objekte, z.B. Rückgaben von `a+b`:
  - ➧ Man kann die Anzahl und die Kosten solcher Objekte verringern.
  - ➧ Sie finden Verweise im Anhang.
- Templates:
  - ➧ Später sprechen wir Strategien an, die Vererbung und `void*`-Zeiger benutzen, um aufgeblähten Codes zu vermeiden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Übliche Fragen

Warum sind "hello world" Programme in C++ so groß im Vergleich zu C?

- iostream vs. stdio
- "hello world" ist kein typisches Programm:
  - ➧ Für kleine Programme können C++ Entwickler stdio noch nutzen.

Warum finden C Entwickler, die auf C++ umsteigen, dass ihr Code oft überraschend groß und langsam ist?

- C++ ist nicht C, und C Entwickler sind keine C++ Entwickler.
- C++ von guten C++ Entwicklern ist genauso gut wie C von guten C Entwicklern.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Effizienz jenseits von C

C++ kann *effizienter* als C sein:

- Die C++ Mechanismen sind oft besser als C Approximationen:
  - ➡ z.B. virtuelle Funktionen
- Abstraktion + Kapselung ⇒ Flexibilität, Implementierungen zu verbessern:
  - ➡ std::strings können leistungsfähiger als char*- basierte Strings sein:
    - ◆ Dürfen "die kleine String Optimierung" (SSO) verwenden
- STL-etablierte Strategien haben Bibliotheken-Design revolutioniert:
  - ➡ Verschieben der Arbeit von Laufzeit hin zur Kompilierzeit:
    - ◆ Template-Metaprogrammierung (TMP), z.B. "traits"
    - ◆ inlined operator()s
  - ➡ Erfolgsbeispiel: sort in C++ ist schneller als qsort in C.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Zusammenfassung: C++ Implementierung

- C++ wurde entworfen, fast immer so klein und schnell wie C zu sein.
- Compiler-erzeugte Datenstrukturen sind normalerweise besser als vergleichbarer handgeschriebener C Code.
- Normalerweise bezahlt man nichts für Features, die man nicht benutzt.
- C++ wird mit Erfolg in vielen Embedded Systems verwendet, z.B.:
  - ➡ Mobile Geräte (z.B. Handys, Smartphones)
  - ➡ Luft- und Raumfahrt
  - ➡ Medizinische Geräte
  - ➡ Videospielkonsolen
  - ➡ Netzwerk- und Telekommunikationshardware (z.B. Routers, usw.)
  - ➡ Navigationsysteme für Frachtschiffe


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# Überblick

Tag 1 (voraussichtlich):

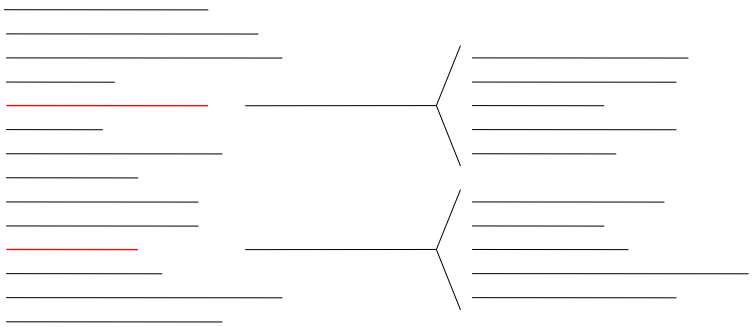
- "Embedded Systems"
- Ein tiefer Blick in C++
  - ➧ Implementierung von Features der Sprache
  - ➧ Inline-Funktionen
  - ➧ Vermeiden von "Code Bloat"
- 3 Ansätze für Interface-basierte Programmierung
- Dynamische Speicherverwaltung
- C++ und ROM (falls die Zeit reicht)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# Vor- und Nachteile der Inline-Funktionen

Vorteile:

- Aufwand für Funktionsaufruf wird eliminiert:
  - ➧ Sehr kurze Funktionen ⇒ gesamte Codegröße könnte abnehmen!
  - ➧ Erforderlich für gute Leistung in geschichteten Systemen.
- Ermöglicht modularen Quellcode mit verzweigungslosem Objektcode.
  - ➧ Funktionsaufrufe im Quellcode erzeugen nur Blöcke im Objektcode.
- Compiler können besser optimieren:




Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Vor- und Nachteile der Inline-Funktionen

Nachteile:

- Debugger haben Probleme:
  - ➧ Wie setzt man einen Breakpoint in einer Funktion, die es nicht gibt?
- Gesamte Codegröße nimmt normalerweise zu.
  - ➧ Das kann die Cache-Hit-Rate reduzieren oder die Seitenwechselquote erhöhen.
- Sie beschränken binäre Kompatibilität für neue Software-Versionen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Vor- und Nachteile der Inline-Funktionen

- "Kleine" Funktionen können zu überraschend viel Code führen:
  - ➧ Aufwand zur Unterstützung von Exceptions könnte bedeutend sein.
  - ➧ Konstruktoren setzen vptrs, rufen Konstruktoren in Basisklassen auf, usw.

```
class Base {
  T1 x, y;
  ...
};
class Derived: public Base {
  T2 z;
public:
  Derived(){}
  ...
};
```

1. Falls auf dem Heap, operator new aufrufen
2. Base::Base aufrufen
3. vptr auf Derived vtbl zeigen lassen
4. Konstruktor für z aufrufen


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Vor- und Nachteile der Inline-Funktionen

inline ist bloß eine Empfehlung — Compiler dürfen sie ignorieren:

- Compiler inlinen virtuelle Funktionsaufrufe nur selten:
  - ➧ Das Inlinen findet im Build-Schritt statt, aber virtuelle Aufrufe werden erst zur Laufzeit aufgelöst.
  - ➧ Optimierungen sind manchmal möglich:
    - ◆ Aufrufe durch Objekte (d.h., nicht durch Zeiger oder Referenzen).
    - ◆ Qualifizierte Aufrufe (d.h., Klassenname::Funktion()).
- Oft ignorieren Compiler "komplexe" Funktionen, z.B. die, die Schleifen enthalten.
- Compiler müssen inline ignorieren, wenn sie einen Zeiger auf eine Funktion brauchen.
  - ➧ z.B. für Konstruktoren/Destruktoren für Arrays von Objekten.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Vor- und Nachteile der Inline-Funktionen

Fazit:

- Fast immer gut, kleine, oft aufgerufene Funktionen zu inlinen.
  - ➧ Hohe Wahrscheinlichkeit, dass das Programm schneller läuft.
- Unbedachtes Inlinen kann zu Code Bloat führen.
- Nutzen Sie Inline-Funktionen sparsam, wenn binäre Kompatibilität wichtig ist.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Überblick

Tag 1 (voraussichtlich):

- "Embedded Systems"
- Ein tiefer Blick in C++
  - ➧ Implementierung von Features der Sprache
  - ➧ Inline-Funktionen
  - ➧ Vermeiden von "Code Bloat"
- 3 Ansätze für Interface-basierte Programmierung
- Dynamische Speicherverwaltung
- C++ und ROM (falls die Zeit reicht)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Code Bloat in C++

Einige C++ Features können aufwändig bezüglich Zeit oder Größe sein, auch wenn sie nicht verwendet werden:

- Unterstützung für Exceptions.
- Unterstützung für erweiterbare iostreams.
  - ➧ d.h. streams, die Typen ausser `char` or `wchar_t` enthalten.

Man könnte so etwas als Bloat betrachten.

Mögliche Abhilfen:

- Exceptions während der Kompilierung deaktivieren.
  - ➧ Praktisch nur, wenn sicher ist, dass kein Code (einschließlich Bibliotheken, Ergänzungen, usw.) wirft.
- auf stdio zugreifen anstatt iostreams.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Code Bloat in C++

Die meisten Bloat-Vorwürfe sind aber unfair und stammen aus:

- der Vergleich von ungleichen Funktionalitäten in C++ und C:
  - ➡ z.B. virtuelle Funktionen in C++ sind leistungsfähiger als C Funktionen.
- missbräuchliche Verwendung der Sprache:
  - ➡ z.B. unnötigen Code in Konstruktoren/Destruktoren stellen.

Das am häufigsten mit Bloat verbundene Sprachelement ist Templates:

- deshalb jetzt der Schwerpunkt Templates.
- "Bloat-Probleme" mit Templates entstehen oft aus:
  - ➡ Missverständnis der Template-Regeln.
  - ➡ falsche Verwendung von Templates.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Templates, Header Dateien und Inline-Funktionen

Beispiel:

```
template<typename T>                    // Header Datei für ein
class SomeClass {                       // Klassen-Template
public:
  SomeClass() { ... }                   // implizit inline deklariert
  void mf1() { ... }                    // implizit inline deklariert
  void mf2();                           // nicht inline deklariert
  ...
};

template<typename T>                    // Funktionstemplates sind
void SomeClass<T>::mf2() { ... }        // i.d.R. in Headern definiert, aber
                                        // das bedeutet nicht, dass sie
                                        // automatisch inline deklariert sind
```

Wichtig:

- Template-Funktionen sollten nicht inline deklariert werden, nur weil sie in Headern definiert sind.
  - ➡ Unnötiges Inlinen verursacht Bloat.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Templates, Header Dateien und Inline-Funktionen

Templates müssen nicht in Headern definiert werden:

```
template<typename T>
class SomeClass {
public:
  SomeClass() { ... }          // noch implizit inline deklariert
  void mf1() { ... }           // noch implizit inline deklariert
  void mf2();                  // nur Deklaration; keine Definition
   ...                         //   ist in dieser Datei
};
```

Code mit diesem Header kompiliert ohne Probleme.

- Aber, wenn SomeClass::mf2 aufgerufen wird, scheitert das Linken.
  ➧ Bald besprechen wir, wie man das löst.
- Templates sind gewöhnlich in Headern, um dieses Problem zu vermeiden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Templates, Header Dateien und Inline-Funktionen

Die Vereinbarung, Template-Code in Header zu stellen, hat einen Vorteil:

- Ein einziger Ort um Code zu warten :
  ➧ Nicht nötig, Header- *und* Implementierungsdateien zu bearbeiten.

Es gibt auch Nachteile:

- steigende Kompilierungszeiten.
- zunehmende Kompilierungs-Abhängigkeiten für Clients.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Template-Instantiierung

Unbenutzte Templates werden nicht instantiiert.

- Sie erzeugen also keinen ausführbaren Code und keine Daten.
  - ➧ Der Compiler muss sie aber lesen; das verlangsamt Kompilierung.
- Templates können deshalb *weniger* Code als nicht-Templates erzeugen!

```
class C {                    // Auch wenn C unbenutzt ist, enthalten
public:                      // Objektdateien normalerweise f1..fn.
  void f1();                 // Nur selten versuchen Linker, unnötige
  ...                        // Code und Daten auszulassen.
  void fn();
};

template<typename T>         // Objektdateien sollen nur die Funktionen
class C {                    // enthalten, die aufgerufen werden.
public:
  void f1();
  ...
  void fn();
};
```

  - ➧ Templates können helfen, das Einbinden von totem Code zu vermeiden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Template-Instantiierung

Instantiierte Templates dürfen sowohl Code als auch Daten erstellen:

```
SomeClass<int> sc;           // SomeClass<int> wird instantiiert;
                             // Code wird erzeugt, Speicher für
                             // statische Klassendaten wird reserviert
```

Die Instantiierung einer Klasse sollte nicht alle Member-Funktionen instantiieren:

- Nur benutzte Member-Funktionen sollten instantiiert werden.
  - ➧ Man sollte nur dafür bezahlen, was man benutzt.
- Einige Compiler (normalerweise ältere) machen das falsch.
  - ➧ Sie instantiieren alle Member-Funktionen einer Klassentemplate, auch wenn nur eine benutzt wird.
  - ➧ Bald sehen wir, wie das verhindert werden kann.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Template-Instantiierung

Templates werden normalerweise *implizit instantiiert*:

- Compiler erkennt benutzte Funktionen und instantiiert sie automatisch.
- Um sie zu erstellen, muss er Zugriff auf ihre Definitionen haben.
  ➡ Deswegen ist Template-Code in Header-Dateien gebräuchlich.
- Ohne Definition erzeugen Compiler Referenz auf ein externes Symbol.
  ➡ SomeClass::mf2 ist also aufrufbar ohne eine Definition, aber während des Linkens taucht eine Fehlermeldung auf.

Templates können auch *explizit instantiiert* werden:

- Die Instantiierung eines Klassen- oder Funktionstemplate fordern.
  ➡ Für Klassen-Templates werden *alle* Member-Funktionen instantiiert.
  ➡ Einzelne Member-Funktionen können auch instantiiert werden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Explizite Instantiierung

In einer .h Datei:

```
template<typename T>          // wie zuvor
class SomeClass {
public:
  SomeClass() { ... }
  void mf1() { ... }
  void mf2();
  ...
};
```

In einer .cpp Datei:

```
...                           // Definitionen für die nicht-inline
                              // Funktionen in SomeClass

template                      // explizite Instantiierung aller SomeClass
class SomeClass<double>;      // Mem-Funktionen für T=double;
                              // erzeugter Code wird in die .obj Datei
                              // dieser .cpp Datei gestellt

template                      // explizite Instantiierung für SomeClass::mf2
void SomeClass<int>::mf2();   // mit T=int; erzeugter Code wird in die .obj
                              // Datei dieser .cpp Datei gestellt
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# Explizite Instantiierung

Explizite Instantiierung kann mühsam sein:

- Man muss alle gewünschten Kombinationen von Templates und Instantiierungsparametern per Hand listen.

Es kann aber nützlich sein:

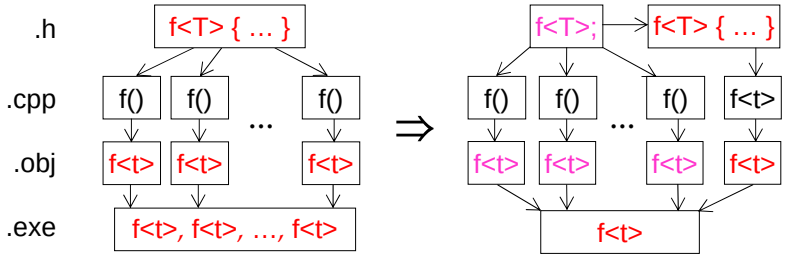
- um Bibliotheken von Instantiierungen zu schaffen.
- um Instantiierungen in bestimmten Code-Sektionen anzulegen.
- um Code Bloat von schlechten Compilern/Linkern zu vermeiden.
  ➧ Details kommen gleich.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# Explizite Instantiierung

Es könnte passieren, dass lauffähige Software mehrfache Instantiierungen enthält:

- falls der Compiler (fälschlicherweise) alle Member-Funktionen einer Klasse instantiiert, wenn nur einige gebraucht werden.
- falls der Linker schlecht ist:




- falls dynamisches Linken benutzt wird.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Das Vermeiden von dupliziertem Code

Gegeben:

```
class IntClass {
public:
   void usageInfo(std::ostream& s);          // gibt Zusammenfassung
   ...                                       // auf s aus
};

class DoubleClass {
public:
   void usageInfo(std::ostream& s);          // gibt Zusammenfassung
   ...                                       // auf s aus
};
```

Beide usageInfo Funktionen haben gleiches Verhalten.

- Das ist duplizierter Code.
- Es verursacht Code Bloat.

Merke: Es gibt keine Templates im Beispiel.



## Das Vermeiden von dupliziertem Code

Eine übliche Weise, solche Vervielfachung zu vermeiden, ist den vervielfachten Code in eine Basisklasse zu schieben:

```
class Base {
public:
   void usageInfo(std::ostream& s);          // gibt Zusammenfassung
   ...                                       // auf s aus
protected:
   ...
};

class IntClass: public Base {
   ...                                       // keine Deklaration für
};                                           // usageInfo

class DoubleClass: public Base {
   ...                                       // keine Deklaration für
};                                           // usageInfo
```

Jetzt gibt es nur eine Kopie von usageInfo im Programm, egal wie viele Klassen von Base erben.




© 2011 Scott Meyers, alle Rechte vorbehalten.

## Das Vermeiden von dupliziertem Code

Genau die gleiche Idee gilt für Templates:

```
template<typename T>                    // ein Template, das Bloat
class SomeClass {                       // verursachen kann
  ...
  void usageInfo(std::ostream& s);      // führt zur Vervielfachung,
  ...                                   // falls Funktion T nicht benutzt
};
```

Die Lösung ist gleich:

```
class Base {                            // wie auf der vorherigen Folie
public:
  void usageInfo(std::ostream& s);
  ...
};
template<typename T>                    // ein Template, das Bloat vermeidet
class SomeClass: public Base {
  ...                                   // keine Deklaration für
};                                      // usageInfo
```



## Das Vermeiden von dupliziertem Code

Das Hieven solches typunabhängigen Codes in Basisklassen ist für Zeiger-Typen besonders hilfreich:

```
template<typename T>                    // allgemeines Template
class Stack { ... };

class GenericPtrStack { ... };          // nicht Template-Klasse, die
                                        // void*s benutzen

template<typename T>                    // partielle Spezialisierung für
class Stack<T*>:                        // Zeiger; benutzt void*-basierte
  private GenericPtrStack {             // Basisklasse für die wirkliche Arbeit
  ...                                   // nur Inline-Funktionen die casten;
};                                      // sie erstellen keinen Code
```

Alle Stack Instantiierungen für Zeiger teilen jetzt ihren Code.

- Später betrachten wir alle Details dieses Beispiels.




© 2011 Scott Meyers, alle Rechte vorbehalten.

## Das Vermeiden von dupliziertem Code

Zusammen können das Hieven von Code und das Inlinen von Funktionen helfen, duplizierten Code durch nicht-Typ Parameter zu vermeiden:

```
template<typename T, std::size_t BUFSZ>  // Verdächtiger Entwurf: jeder Wert
class Buffer {                           // für BUFSZ erzeugt eine neue
  T buffer[BUFSZ];                       // Gruppe von Member-Funktionen
public:
  ...
};

template<typename T>                     // Besserer Entwurf: BufferBase
class BufferBase {                       // ist unabhängig von BUFSZ
  ...
};
template<typename T, std::size_t BUFSZ>  // Buffer macht nur BUFSZ-
class Buffer: public BufferBase<T> {     // unabhängige Operationen.
  ...                                    // Idealerweise sind alle Inline-
};                                       // Funktionen ⇒ Buffer-Klassen
                                         // kosten nichts
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Das Vermeiden von dupliziertem Code

Die Vermeidung von Template-basierter Codeduplizierung benötigt *Gemeinsamkeits- und Variabilitätsanalyse*:

- Die Teile des Templates, welche von den Template-Parametern nicht abhängig sind (die *gemeinsamen* Teile), sollten raus aus dem Template.
- Die übrigen Teile (die *variablen* Teile) sollten im Template bleiben.

Solche Analyse ist wichtig, um Codeduplizierung generell zu vermeiden:

- Features, die Klassen gemeinsam haben, sollten von den Klassen entfernt werden.
  - ➧ z.B. in eine Basisklasse.
  - ➧ z.B. in ein Klassen-Template.
- Features, die Funktionen gemeinsam haben, sollten von den Funktionen entfernt werden.
  - ➧ z.B. in eine neue Funktion.
  - ➧ z.B. in ein Funktionstemplate.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Code Bloat Zusammenfassung

Guter Entwurf kann den meisten Bloat vermeiden. Mögliche Ansätze:

- Compiler-Unterstützung für Exceptions ausschalten.
- stdio anstatt iostreams benutzen.
- sorgfältig inlinen, besonders bei Templates.
- explizite anstatt implizite Instantiierung verwenden.
- Parameter-unabhängigen Code aus Templates entfernen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Der Fall Funktionstemplates

Der Fall Funktionstemplates ist analog zu Klassen-Templates:

```
template<typename T>                // Template, das zu Bloat führen könnte
void doSomething(const T& obj)
{
  ...                               // Code, der T oder obj benutzt
  ...                               // Code, der von T und obj unabhängig ist
  ...                               // Code, der T oder obj benutzt
}
```

Eine Alternative:

```
void doSomethingHelper();           // Typ-unabhängiger Code in nicht-
                                    // Template Funktion; nicht inline

template<typename T>                // geändertes Template, das zu Bloat
void doSomething(const T& obj)      // führt
{
  ...                               // Code, der T oder obj benutzt
  doSomethingHelper();
  ...                               // Code, der T oder obj benutzt
}
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Daten Bloat

Nicht aller Bloat stammt aus Code. Unnötige Klassen können auch dazu beitragen:

- Einige Klassen haben vtbls: Unnötige Klassen ⇒ unnötige vtbls.
  - ➧ Die Klassen könnten von Templates erzeugt werden.
- Exceptions müssen richtig behandelt werden: Unnötige nicht-inline Funktionen ⇒ unnötige Tabellen für Ausnahmebehandlung.
  - ➧ Die Funktionen könnten von Templates erzeugt werden.
  - ➧ Gültig nur, wenn der Tabellen-Ansatz für Exceptions benutzt wird.

Diese Sorgen betreffen keine Klassen-Templates, die:

- nur Inline-Funktionen enthalten.
  - ➧ Sie erzeugen deswegen keine Ausnahmentabellen.
- keine virtuellen Funktionen enthalten.
  - ➧ Sie haben deswegen keine vtbl.

Später sehen wir Beispiele von solchen "Bloat-freien" Templates.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Überblick

Tag 1 (voraussichtlich):

- "Embedded Systems"
- Ein tiefer Blick in C++
  - ➧ Implementierung von Features der Sprache
  - ➧ Inline-Funktionen
  - ➧ Vermeiden von "Code Bloat"
- 3 Ansätze für Interface-basierte Programmierung
- Dynamische Speicherverwaltung
- C++ und ROM (falls die Zeit reicht)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Interface-basierte Programmierung

*Interface-basierte Programmierung:*

- Codieren gegen ein Interface, das mehrere Implementierungen ermöglicht.
  - ➧ Interface für eine Funktion.
  - ➧ Interface für eine Klasse.
- Code der Clients weiß nicht, welche Implementierung er benutzt.
  - ➧ Er ist nur auf das Interface angewiesen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Polymorphismus

*Polymorphismus:*

- Die Verwendung verschiedener Implementierungen durch ein einzelnes Interface.

Schlüsselfrage: **Wann wird bekannt, welche Implementierung benutzt werden soll?**

- **Zur Laufzeit:** jeder *Aufruf* darf eine verschiedene Implementierung nutzen.
  - ➧ Verwenden Sie Vererbung + virtuelle Funktionen.
- **Zur Linkzeit:** jedes *Linken* darf einen neuen Satz Implementierungen erzeugen.
  - ➧ Verwenden Sie unabhängig kompilierte Funktionsrümpfe.
  - ➧ Gültig sowohl für statisches als auch dynamisches Linken.
- **Zur Kompilierzeit:** jedes *Kompilieren* darf einen neuen Satz Implementierungen erzeugen.
  - ➧ Verwenden Sie errechnete typedefs.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Polymorphismus zur Laufzeit

- Die "normale" Bedeutung von Interface-basierter Programmierung.
  - ➧ In vieler OO Literatur die einzige Bedeutung.
    - ◆ Unnötig beschränkt für C++.
- Am flexibelsten.
  - ➧ Kann erst zur Laufzeit bekannte Informationen ausnutzen.
- Am aufwändigsten.
  - ➧ Basiert auf vptrs, vtbls, nicht-inlinen Funktionsaufrufen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Beispiel von Polymorphismus zur Laufzeit

```
class Packet {                                  // Basisklasse
public:                                         // ("Interface")
  …
  virtual bool isWellFormed() const = 0;
  virtual std::string payload() const = 0;
  …
};
class TCPPacket: public Packet {                // Abgeleitete Klasse
  …                                             // ("Implementierung")
  virtual bool isWellFormed() const;
  virtual std::string payload() const;
  …
};
class CANPacket: public Packet {                // Abgeleitete Klasse
  …                                             // ("Implementierung")
  virtual bool isWellFormed() const;
  virtual std::string payload() const;
  …
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beispiel von Polymorphismus zur Laufzeit

```
std::auto_ptr<Packet>                    // Fabrikfunktion; erzeugt
nextPacket( /* params */ );              // nächstes Paket

…

std::auto_ptr<Packet> p;
while (p = nextPacket( /* params */), p.get() != 0) {
  if (p->isWellFormed()) {               // Packet Interface nutzen

    …

  }

  …
}
```

Polymorphismus zur Laufzeit ist hier sinnvoll:

- Arten von Paketen sind zur Laufzeit unterschiedlich.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Polymorphismus zur Linkzeit

- Nützlich, wenn Information zur Linkzeit bekannt ist, aber nicht zur Kompilierzeit.
- Keine Notwendigkeit, virtuelle Funktionen zu verwenden.
- Schaltet Inline-Funktionen normalerweise aus.
  ➧ Das Inlinen findet in der Regel während des Kompilierens statt.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beispiel von Polymorphismus zur Linkzeit

Software kann auf zwei Arten von Geräten eingesetzt werden:

- teures, leistungsstarkes Gerät.
  - ➧ Verwendet teure, leistungsstarke Bauteile.
- billigeres Gerät, das weniger Leistung bietet.
  - ➧ Verwendet billigere Bauteile, die weniger Leistung bieten.
- Im wesentlichen läuft die gleiche Software auf beiden Geräten.
  - ➧ Implementierungen der Gerätetreiber sind unterschiedlich.
    - ◆ Ein gemeinsames Interface kann definiert werden.

Ansatz:

- eine einzige Klassendefinition für beide Treiber.
- zwei geräteabhängige Implementierungen.
- Implementierung wird durch das Linken ausgewählt.
  - ➧ Das ist "C" Polymorphismus.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Beispiel von Polymorphismus zur Linkzeit

device.h:

```
namespace Drivers {
  class Impl;
  class DeviceDriver {          // nur nicht-virtuelle nicht-inline Funktionen
  public:
    DeviceDriver();
    ~DeviceDriver();
    void reset();
    …
  private:
    Impl *pImpl;                // Zeiger auf Daten für Treiber
  };
}
```

Alle Clients #include diesen Header und codieren gegen diese Klasse.

- Merke: Keine virtuelle Funktionen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beispiel von Polymorphismus zur Linkzeit

EFDevice.cpp (erzeugt EFDevice.o, EFDevice.obj oder EFDevice.dll usw.):

- EFDevice = "Expensive Fast Device"

```
namespace Drivers {
  struct Impl { … };                    // Daten für EFDevice Treiber
  DeviceDriver::DeviceDriver()          // Ctor Code für EFDevice
  { … }
  DeviceDriver::~DeviceDriver()         // Dtor Code für EFDevice
  { … }
  void DeviceDriver::reset()            // zurücksetzender Code
  { … }                                 // für EFDevice
  …
}
```

Alle Funktionen in dieser Datei haben Zugriff auf die Impl struct, die hier definiert ist.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Beispiel von Polymorphismus zur Linkzeit

CSDevice.cpp (erzeugt CSDevice.o, CSDevice.obj, oder CSDevice.dll, usw.):

- CSDevice = "Cheap Slow Device"

```
namespace Drivers {
  struct Impl { … };                    // Daten für CSDevice Treiber
  DeviceDriver::DeviceDriver()          // Ctor Code für CSDevice
  { … }
  DeviceDriver::~DeviceDriver()         // Dtor Code für CSDevice
  { … }
  void DeviceDriver::reset()            // zurücksetzender Code
  { … }                                 // für CSDevice
  …
}
```

Alle Funktionen in dieser Datei haben Zugriff auf diese Impl struct.

- Impl in dieser Datei normalerweise anders als Impl in EFDevice.cpp.
- Funktionsrümpfe in dieser Datei normalerweise auch anders.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beispiel von Polymorphismus zur Linkzeit

Man linkt mit:

- EFDevice.o, wenn man für ein teures, leistungsstarkes Gerät baut.
  - ➧ Oder man linkt dynamisch mit z.B. EFDevice.dll.
- CSDevice.o, wenn man für ein billigeres, weniger leistungsstarkes Gerät baut.
  - ➧ Oder dynamisch mit z.B. CSDevice.dll.

Polymorphismus zur Linkzeit ist hier sinnvoll:

- Einsatzplattform unbekannt während des Kompilierens, aber bekannt während des Linkens.
  - ➧ Kein Bedarf an Flexibilität oder Aufwand von Laufzeit-Polymorphismus.
    - Keine vtbls.
    - Keine Indirektion über vtbls.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Polymorphismus zur Kompilierzeit

- Nützlich wenn
  - ➧ möglich ist, die Implementierung während des Kompilierens festzustellen.
  - ➧ man größtenteils Implementierungs-unabhängigen Code schreiben will.
- Kein Bedarf an virtuellen Funktionen.
- Ermöglicht das Inlinen.
- Basiert auf *impliziten Interfaces*.
  - ➧ Die vorher gezeigten Arten von Polymorphismus sind auf expliziten Interfaces basiert.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Noch einmal das Beispiel des Gerätetreibers

Ziel:

- **Die richtige Geräteklasse aus der Zahl der Bits/Zeiger zu bestimmen.**
  - ➧ Das ist während des Kompilierens bekannt.

Ansatz:

- zwei oder mehr Klassen mit "kompatiblen" Interfaces schreiben.
  - ➧ d.h. sie unterstützen ein gemeinsames Interface.
    - ◆ z.B. müssen eine reset Funktion bieten, die mit keinen Argumenten aufgerufen werden darf.
- Information zur Kompilierzeit verwenden, um die für die Anwendung passende Klasse zu wählen.
- ein typedef für diese Klasse definieren.
- mit diesem typedef programmieren.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Beispiel von Polymorphismus zur Kompilierzeit

Geänderte device.h:

```
#include "NASDevice.h"      // NAS = "Normal Address Space" (32 bits);
                            // definiert class NASDevice
#include "BASDevice.h"      // BAS = "Big Address Space" (>32 bits);
                            // definiert class BASDevice
#include "SASDevice.h"      // SAS = "Small Address Space" (<32 bits);
                            // definiert class SASDevice
...                         // der Rest von device.h (kommt bald)
```

Nach Entwurf bietet jede Klasse ein kompatibles Interface an.

- Members mit gleichen Namen, kompatiblen Typen, usw.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beispiel von Polymorphismus zur Kompilierzeit

Treiberklassen dürfen alle Sprachelemente benutzen:

- **vornehmlich Inline-Funktionen.**

```
class NASDevice {
public:
  …
  void reset() { … }                    // Inline-Funktion
  …
};
class BASDevice {
public:
  …
  void reset() { … }                    // Inline-Funktion
  …
};
class SASDevice {
  …
  void reset();                         // Nicht-Inline-Funktion
  …
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Beispiel von Polymorphismus zur Kompilierzeit

Clients greifen auf den richtigen Treibertyp so zu:

```
Driver::type d;                 // Der Typ von d ist entweder NASDevice,
d.reset();                      // BASDevice oder SASDevice,
                                // je nach der Zahl der Bits/Zeiger
```

- Driver "berechnet" die richtige Klasse, auf die type verweist.
  - ➡ Implementierung auf der nächsten Folie.

Polymorphismus zur Kompilierzeit ist hier sinnvoll:

- Typ des Geräts kann während des Kompilierens festgestellt werden.
  - ➡ Kein Bedarf an Flexibilität oder Aufwand von Laufzeit-Polymorphismus.
  - ➡ Kein Bedarf an Einstellung vom Linker oder Verzicht auf Inlinen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beispiel von Polymorphismus zur Kompilierzeit

Geänderte device.h (fortgesetzt):

```
template<int PtrBitsVs32> struct DriverChoice;
template<> struct DriverChoice<-1> {                    // Wenn Bits/Zeiger < 32
   typedef SASDevice type;
};
template<> struct DriverChoice<0> {                     // Wenn Bits/Zeiger == 32
   typedef NASDevice type;
};
template<> struct DriverChoice<1> {                     // Wenn Bits/Zeiger > 32
   typedef BASDevice type;
};


struct Driver {
  enum { bitsPerVoidPtr = CHAR_BIT * sizeof(void*) };
  enum { ptrBitsVs32 = bitsPerVoidPtr >    32  ? 1:
                       bitsPerVoidPtr ==   32  ? 0:
                                                 -1
       };
  typedef DriverChoice<ptrBitsVs32>::type type;
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Zusammenfassung: Interface-basierte Programmierung

- Ein Interface, mehrfache Implementierungen.
- Polymorphismus wird benutzt, die Implementierung auszuwählen.
  - ➧ Polymorphismus zur Laufzeit benutzt virtuelle Funktionen.
  - ➧ Polymorphismus zur Linkzeit benutzt Linkereinstellung.
  - ➧ Polymorphismus zur Kompilierzeit benutzt typedefs.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# Überblick

Tag 1 (voraussichtlich):

- "Embedded Systems"
- Ein tiefer Blick in C++
  - ➡ Implementierung von Features der Sprache
  - ➡ Inline-Funktionen
  - ➡ Vermeiden von "Code Bloat"
- 3 Ansätze für Interface-basierte Programmierung
- Dynamische Speicherverwaltung
- C++ und ROM (falls die Zeit reicht)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# Dynamische Speicherverwaltung

Embedded Entwickler sagen oft, dass Speicherverwaltung kein Thema ist:

- Kunde: "Wir haben keinen Heap."
- Ich: "Nein. Sie haben fünf Heaps."

**Dynamische Speicherverwaltung oft ein Teil von Embedded Systems.**

- Auch wenn niemand malloc/free/new/delete aufruft.
- Hauptindiz:
  - ➡ Objekte variabler Grösse landen in einem Speicher fester Grösse.
    - ◆ z.B. Event/Error logs, Emails, usw.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.

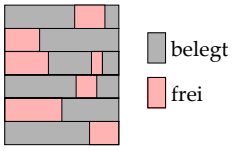


© 2011 Scott Meyers, alle Rechte vorbehalten.

## Dynamische Speicherverwaltung

Vier der üblichsten Bedenken:

- **Geschwindigkeit:**
  - ➧ Sind new/delete/malloc/free schnell genug?
  - ➧ Mit welcher Varianz, d.h. wie deterministisch?
- **Fragmentierung:**
  - ➧ Wird der Heap in zu kleine Blöcke zersetzt?
    - ◆ Das ist *externe* Fragmentierung.
- **Speicherlecks:**
  - ➧ Werden einige Allokationen nicht dealloziert?
- **Speichermangel:**
  - ➧ Was ist, wenn eine Allokation fehlschlägt?




Jede dieser Bedenken kann man angehen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Überblick über Allokationsstrategien

Jede davon nicht so allgemein wie malloc/free/new/delete.

- Besonders passend im embedded Bereich.

Was wir untersuchen:

- Rein statische Allokation
- LIFO Allokation
- Pool Allokation
- Block Allokation
- Bereichsallokation
  - ➧ Eine Optimierung, die mit anderen Strategien zusammen eingesetzt werden kann.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Rein statische Allokation

Kein Heap. Objekte sind entweder

- auf dem Stack: lokal in einer Funktion.
- mit statischer Speicherdauer:
  - im **globalen Scope**.
  - im **Scope eines Namensraums**.
  - **static in einer Datei, Funktion oder Klasse.**

Nützlich wenn:

- exakte oder maximale Anzahl benötigter Objekte im Voraus bekannt.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Rein statische Allokation

"Allokation" geschieht beim Bauen. Deswegen:

- **Geschwindigkeit**: unendlich; deterministisch.
- **externe Fragmentierung**: unmöglich.
- **Speicherlecks**: unmöglich.
- **Speichermangel**: unmöglich.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## "Heap Allokation"

Zwei gebräuchliche Bedeutungen:

- **dynamische Allokation außerhalb vom Laufzeitsstack.**
- *irreguläre* dynamische Allokation außerhalb vom Laufzeitsstack.
  - ➧ unvorhersehbare Anzahl von Objekten.
  - ➧ unvorhersehbare Größe von Objekten.
  - ➧ unvorhersehbare Lebensdauer von Objekten.

Wir meinen die erste Bedeutung.

- Die zweite ist bloß der allgemeinste (d.h. schwierigste) Fall der ersten.

Heapverwaltung existiert, auch wenn Speicher für Objekte unterschiedlicher Größe manuell verwaltet wird:

```
unsigned char buffer[SomeSize];      // das ist eigentlich ein Heap
…                                    // anlegen/löschen mehrerer
                                     // Objekte mit unterschiedlichen
                                     // Größen im buffer
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Der C++ Ansatz zur Speicherverwaltung

Benutzer-definierte Speicherverwaltung basiert typischerweise auf:

- Benutzerdefinierten Implementierungen von malloc/free
- Benutzerdefinierten Implementierungen von operator new/new[] und operator delete/delete[]
- New-Handler:
  - ➧ Funktionen, die aufgerufen werden, sobald operator new/new[] fehlschlagen.

Details finden Sie in den weiterführenden Informationen.

- Hier besprechen wir Ansätze, die besonders für Embedded Systems passen.

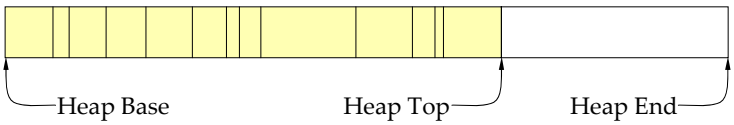

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# LIFO-Heap Allokation

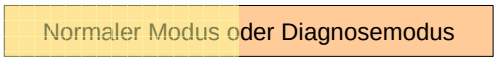
Dynamische Allokation muss LIFO sein (wie ein Stack).




Ermöglicht die Umsetzung einer "Union" verschiedener Betriebsmodi:

- z.B., ein System im Modus "Normal" oder "Diagnose".
  - ➧ Statische Allokation benötigt die *Summe* des Speicherbedarfs.


  - ➧ LIFO Allokation erfordert nur das *Maximum*.





Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# LIFO-Heap Allokation

Erster Entwurf:

```
class LIFOAllocator {                                   // bietet Verhalten
public:                                                 // von new/delete via
  LIFOAllocator(unsigned char* heapAddr,                // allocate/deallocate
                std::size_t heapSize)
  : heapBase(heapAddr), heapEnd(heapAddr+heapSize),
    heapTop(heapAddr)
  {}
  void* allocate(std::size_t sz) throw (std::bad_alloc);   // kommt bald
  void deallocate(void* ptr, std::size_t sz) throw ();     // kommt auch bald
private:
  unsigned char * const heapBase;
  unsigned char * const heapEnd;
  unsigned char *heapTop;
};
```

- allocate/deallocate verhalten sich wie klassenspezifische new/delete.
- Zeiger als Datenelement ⇒ Kopierfunktionen soll deklariert werden.
- LIFOAllocator als Template ⇒ Ctor-Param. werden Template-Param.
  - ➧ Abschnitt MMIO zeigt ein Beispiel.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## LIFO-Heap Allokation

Auf LIFOAllocator können Klassen maßgeschneiderte new/delete leicht bauen:

```
unsigned char heapSpace[HeapSpaceSize];        // Speicher für Heap

LIFOAllocator globalAllocator(heapSpace,       // normalerweise im
                              HeapSpaceSize);  // globalen Scope

void* Widget::operator new(std::size_t bytes) throw (std::bad_alloc)
{
  return globalAllocator.allocate(bytes);
}
void Widget::operator delete(void *ptr, std::size_t size) throw ()
{
  globalAllocator.deallocate(ptr, size);
}
```

Hier gibt es einen globalen Heap, aber pro Klasse oder pro Thread ist einfach.

- Einen LIFOAllocator für jeden Speicherblock, der einen LIFO-Heap darstellt.
  - ➧ Allokator pro Klasse: LIFOAllocators sind statisch und privat.
  - ➧ Allokator pro Thread: Thread-Local Storage (TLS) für Speicher.

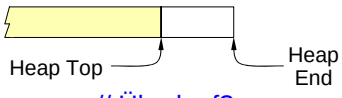

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## LIFOAllocator::allocate

Implementiert wie globaler operator new:

```
void* LIFOAllocator::allocate(std::size_t bytes) throw (std::bad_alloc)
{
  if (bytes == 0) bytes = 1;
  while (true) {
    if (heapTop + bytes <= heapEnd) {          // Überlauf?
      unsigned char *pMem = heapTop;           // Ausrichtung?
      heapTop += bytes;
      return pMem;
    }
    std::new_handler currentHandler = std::set_new_handler(0);
    std::set_new_handler(currentHandler);

    if (currentHandler) currentHandler();
    else throw std::bad_alloc();
  }
}
```




- Kommentare zeigen Probleme, die wir ignorieren.
- Bei diesem Entwurf fällt es dem Handler schwer, den verfügbaren Speicher zu vergrößern.

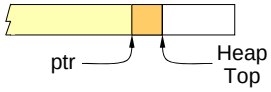

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## LIFOAllocator::deallocate

```
void LIFOAllocator::deallocate(void *ptr, std::size_t size) throw ()
{
  if (ptr == 0) return;
  if (heapTop != static_cast<unsigned char*>(ptr) + size) {
    // Datenstruktur für Heap ist ungültig!
    Problem loggen, dann exit oder abort aufrufen
    oder das System neu starten / rebooten.
  }
  heapTop -= size;
}
```



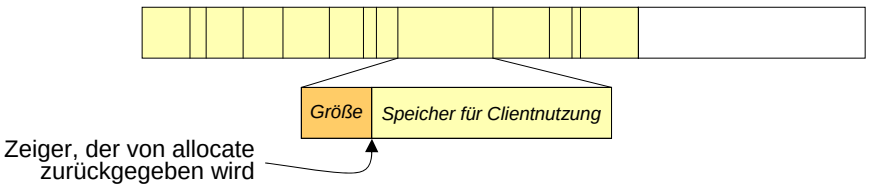

- Exception-Spezifikation ⇒ Keine Exception darf geworfen werden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Globale new/delete

Globalem operator delete fehlt size, aber heapTop -= size nötig, also:

- LIFOAllocator::allocate optional die Größeninfo speichern lassen.




- LIFO::deallocate überladen, um verborgene Größeninfo zu nutzen.

```
class LIFOAllocator {
public:
  …
  void* allocate(std::size_t sz, bool hideSize) throw (std::bad_alloc);
  void deallocate(void* ptr, std::size_t sz) throw ();
  void deallocate(void* ptr) throw ();
};
```

Es wäre besser, einen enum statt eines bool zu verwenden…


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Globale new/delete

Globale new/delete sind dann leicht zu implementieren:

```
void* operator new(std::size_t bytes) throw (std::bad_alloc)
{
  return globalAllocator.allocate(bytes, true);
}
void operator delete(void *ptr) throw ()
{
  globalAllocator.deallocate(ptr);          // Merke: kein "size" param
}
```

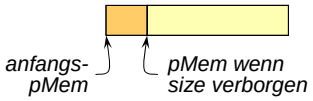
Ebenso wie klassenspezifische Versionen:

```
void* Widget::operator new(std::size_t bytes) throw (std::bad_alloc)
{
  return globalAllocator.allocate(bytes, false);
}
void Widget::operator delete(void *ptr, std::size_t size) throw ()
{
  globalAllocator.deallocate(ptr, size);
}
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Globale new/delete

```
void* LIFOAllocator::allocate(std::size_t bytes, bool hideSize) throw (std::bad_alloc)
{
  if (bytes == 0) bytes = 1;
  if (hideSize)
    bytes += sizeof(std::size_t);   // Platz für size
                                    // hinzufügen; Überlauf?
  while (true) {
    if (heapTop + bytes <= heapEnd) {                    // Überlauf?
      unsigned char *pMem = heapTop;                     // Ausrichtung?
      if (hideSize) {
        *reinterpret_cast<std::size_t*>(pMem) = bytes;   // Ausrichtung?
        pMem += sizeof(std::size_t);                     // Ausrichtung?
      }
      heapTop += bytes;
      return pMem;
    }
    den New-Handler wie üblich prüfen/nutzen;
  }
}
```





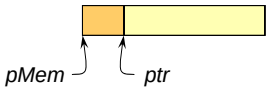
Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Globale new/delete

```
void LIFOAllocator::deallocate(void *ptr) throw ()
{
  if (ptr == 0) return;
  unsigned char *pMem =
    static_cast<unsigned char*>(ptr) – sizeof(std::size_t);
  std::size_t size = *reinterpret_cast<std::size_t*>(pMem);
  if (heapTop != pMem + size) {
    // Datenstruktur für Heap ist ungültig!
    Problem loggen, dann exit oder abort aufrufen
    oder das System neu starten / rebooten.
  }
  heapTop -= size;
}
```





Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## LIFO-Heap Allokation

- **Geschwindigkeit**: extrem schnell; deterministisch.
  ➧ vorausgesetzt, es gibt genügend Speicher.
- **Externe Fragmentierung**: möglich, aber leicht aufspürbar (wie gezeigt).
- **Speicherlecks**: möglich, leicht aufzuspüren.
- **Speichermangel**: möglich.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# Pool Allokation

Alle Allokationen im Heap sind gleich groß.

- in der Regel, weil alle Objekte gleich groß sind.
  - ➧ paßt gut für klassenspezifische Allokatoren.
- nützlich auch, wenn alle Heap Objekte *annähernd* gleich groß sind.
  - ➧ Dann sind alle Allokationen der Grösse der grössten Objekten.

Ansatz:

- Heap-Speicher als Array betrachten.
  - ➧ Jedes Element des Arrays ist eine Allokationseinheit.
    - ◆ Keine Notwendigkeit, die Größe jeder Allokation zu speichern.
- Unbelegte Elemente werden auf einer *Freiliste* gehalten.
- Allokation/Deallokation sind einfache Operationen auf der Liste:
  - ➧ Entfernen/Hinzufügen der Elemente am Anfang der Liste.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# Pool Allokation

```
template<std::size_t ElementSize>
class PoolAllocator {
public:
  PoolAllocator(unsigned char* heapAddr,
                std::size_t heapSize);                  // nächste Folie

  void* allocate(std::size_t sz) throw (std::bad_alloc); // kommt bald
  void deallocate(void* ptr, std::size_t sz) throw ();   // kommt auch bald

private:
  union Node {                                          // Element des Pools
    unsigned char data[ElementSize];                    // falls im Einsatz
    Node *next;                                         // falls frei
  };

  Node *freeList;
};
```

- Zeiger als Datenelement ⇒ Kopierfunktionen sollten deklariert werden.
- Wenn PoolAllocator kein Template wäre, könnte der Template Parameter ein Konstruktor Parameter sein.
- Am besten stellen wir sicher, dass ElementSize > 0.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



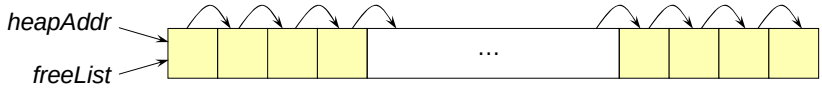
© 2011 Scott Meyers, alle Rechte vorbehalten.

## PoolAllocator Konstruktor

```
template<std::size_t ElementSize>
PoolAllocator<ElementSize>::PoolAllocator( unsigned char* heapAddr,
                                           std::size_t heapSize)
: freeList(reinterpret_cast<Node*>(heapAddr))
{
  const std::size_t nElems = heapSize / ElementSize;

  for (std::size_t i = 0; i < nElems-1; ++i)          // Arrayelemente
    freeList[i].next = &freeList[i+1];                // nacheinander verlinken

  freeList[nElems-1].next = 0;
}
```





Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## PoolAllocator::allocate

```
template<std::size_t ElementSize>
void* PoolAllocator<ElementSize>::allocate(std::size_t bytes)
  throw (std::bad_alloc)
{
  if (bytes != ElementSize) return ::operator new(bytes);

  while (true) {
    if (freeList != 0) {
      void *pMem = freeList;                          // Ausrichtung?
      freeList = freeList->next;
      return pMem;
    }

    std::new_handler currentHandler = std::set_new_handler(0);
    std::set_new_handler(currentHandler);

    if (currentHandler) currentHandler();
    else throw std::bad_alloc();
  }
}
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## PoolAllocator::deallocate

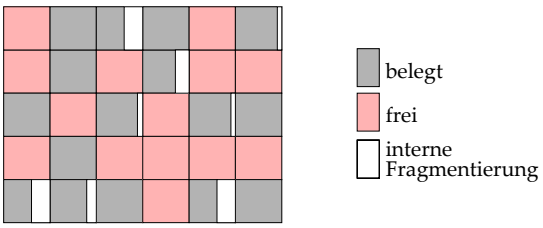
```
template<std::size_t ElementSize>
void PoolAllocator<ElementSize>::deallocate(void *ptr, std::size_t size)
  throw ()
{
  if (ptr == 0) return;
  if (size != ElementSize) {
    ::operator delete(ptr);
    return;
  }
  Node *p = static_cast<Node*>(ptr);
  p->next = freeList;
  freeList = p;
}
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## PoolAllocator::allocate

Variante: erlaube bytes <= ElementSize.

- d.h. die Anforderung muss lediglich hineinpassen.
  - ➧ flexibler, kann aber zu *interner* Fragmentierung führen.





Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Pool Allokation

Code der Clients:

- Clients implementieren Operatoren operator new/delete.
- Ihnen zur Übung überlassen :-)
  - ➡ operator new ruft allocate auf.
  - ➡ operator delete ruft deallocate auf.
  - ➡ ähnlich wie LIFOAllocator.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Pool Allokation

- **Geschwindigkeit**: extrem schnell; deterministisch.
  - ➡ vorausgesetzt:
    - ◆ keine Anforderungen von unerwarteter Größe.
    - ◆ es gibt genügend Speicher.
- **Externe Fragmentierung**: unmöglich.
- **Speicherlecks**: möglich.
- **Speichermangel**: möglich.

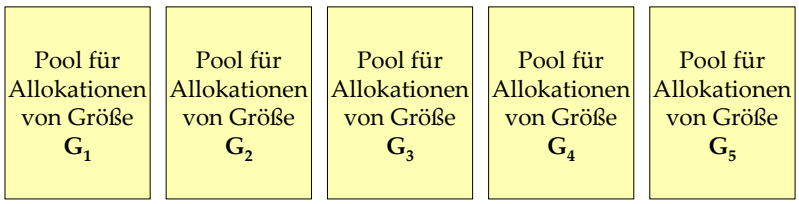

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Block Allokation

Im Grunde ein Satz von Pools mit unterschiedlichen Element-Größen:




Anfrage für $n$ Bytes wird vom ersten Pool mit Größe ≥ $n$ und nicht-null Freiliste behandelt.

Nützlich wenn:

- Allokationen für ziemlich wenig Objektgrößen gebraucht werden.
  - ➧ sonst internale Fragmentierung ⇒ verschwendeter Speicher.

Viele RTOSes bieten unmittelbare Unterstützung für Block-Allokatoren an.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Block Allokation

- **Geschwindigkeit**: schnell; nahezu deterministisch (und begrenzbar).
  - ➧ vorausgesetzt:
    - keine Anforderungen, die zu groß für den Pool der größten Größe $G_n$ sind.
    - es gibt genügend Speicher.
- **Externe Fragmentierung**: unmöglich.
- **Speicherlecks**: möglich.
- **Speichermangel**: möglich.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Allgemeine Allokation unterschiedlicher Größen

Was new/delete/malloc/free schon erledigen.

- Lohnend nur, wenn die Standard-Funktionen unzureichend sind.

Mögliche Motivationen:

- Überschreitungen/Unterschreitungen aufspüren.
- Nutzungsdaten des Heaps sammeln.
  - ➧ Distribution von Größen und Lebensdauern, zeitliche Muster, usw.
- Zusammenlegung von Datenstrukturen ermöglichen.
- Aufwände für Thread-Sicherheit vermeiden.
  - ➧ Anwendungen mit nur einem Thread.
  - ➧ Threadlokale Allokatoren in Multithread-Anwendungen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Bereichsallokation

Eine Optimierung für den Fall, wenn der Speicher aller Objekte in einem Heap gleichzeitig freigegeben werden kann.

- Clients rufen eine Member-Funktion des Bereichs zur passenden Zeit auf.
  - ➧ schneller als den Speicher jedes Objekts einzeln freizugeben.
- Oft mit LIFO-Allokatoren, geht aber auch mit Pools, Blöcken, usw.
- operator delete für einzelne Objekte der Leerbefehl ⇒ sehr schnell.
  - ➧ delete-Operator noch nützlich, um Destruktoren aufzurufen:

```
delete p;       // ruf *p's Dtor auf, dann operator delete auf p.
                // *p in einem Bereich ⇒ operator delete ist Leerbefehl
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Zusammenfassung: dynamische Speicherverwaltung

- Embedded Systems setzen oft dynamische Speicherverwaltung ein.
- Schlüsselthemen sind Geschwindigkeit, Fragmentierung, Lecks und Speichermangel.
- LIFO ist schnell und ohne Fragmentierung, aber Lebensdauer der Objekte muss LIFO sein.
- Pools sind schnell und (möglicherweise) ohne Fragmentierung, aber Größe der Objekte ist beschränkt.
- Block Allokation ist im Grunde mehrfache Pool Allokatoren.
- Bereiche sind überragend, wenn alle Heap-Objekte gleichzeitig freigegeben werden dürfen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Überblick

Tag 1 (voraussichtlich):

- "Embedded Systems"
- Ein tiefer Blick in C++
  - ➧ Implementierung von Features der Sprache
  - ➧ Inline-Funktionen
  - ➧ Vermeiden von "Code Bloat"
- 3 Ansätze für Interface-basierte Programmierung
- Dynamische Speicherverwaltung
- C++ und ROM (falls die Zeit reicht)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C++ und ROM

Alles darf in das ROM geschrieben und in das RAM vor der Programm-Ausführung geladen werden.

- So lange die Architektur es erlaubt.
  - ➧ Harvard ist ein Gegenbeispiel.

Die interessantere Frage ist:

- Was darf im ROM bleiben, während das Programm läuft?

Der C++-Standard schweigt zu diesem Thema:

- Alles ist erlaubt, nichts ist garantiert.
- Die Regeln für ROMbarkeit hängen deshalb vom Compiler und Linker ab.

Im Folgenden spreche ich an, was *technisch möglich* ist.

- Ihre Compiler/Linker haben wahrscheinlich etliche Beschränkungen.
- Zuerst besprechen wir diese Beschränkungen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## C++ und ROM

Um die Beschränkungen zu verstehen, muss man sich mit "POD-Typen" auskennen.

- "POD" = "Plain Old Data" (≅ schlichte Daten).
- Alle Typen in C sind POD-Typen.
- C++ Klassen/Structs/Unions sind POD-Typen, wenn ihnen folgendes fehlt:
  - ➧ Basisklassen
  - ➧ Virtuelle Funktionen
  - ➧ Protected oder private, nicht-statische Datenelemente
  - ➧ Nicht-statische Datenelemente, die Referenzen sind
  - ➧ Benutzerdefinierte Konstruktoren, Destruktoren oder Kopierzuweisungsoperatoren
  - ➧ Nicht-statische Datenelemente, die Nicht-POD-Typen sind

Im wesentlichen ist eine Klasse oder Struct in C++ ein POD-Typ, wenn sie "sich wie C benimmt und so aussieht."

- Beachten Sie, dass nicht-virtuelle Member-Funktionen gestattet sind.
- Genauso wie statische Daten und Member-Funktionen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C++ und ROM

Gebräuchliche Beschränkungen zum Rommen von Daten:

- Oft rommen Compiler/Linker nur statisch initialisierte POD-Typen.
  - ➧ Bald sehen wir, dass es technisch möglich ist, dynamisch initialisierte Nicht-POD-Typen zu rommen.
- Einige Compiler/Linker rommen Structs, aber nicht Klassen.
  - ➧ Es gibt keinen technischen Grund für so eine Unterscheidung.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## C++ und ROM

Programmbefehle können immer in das ROM gestellt werden.

Daten in einem C++ Programm dürfen gerommt werden, falls:

- ihre Werte vor der Laufzeit bekannt sind.
  - ➧ d.h. der Compiler oder Linker kennt sie oder kann sie berechnen.
- sie zur Laufzeit nicht geändert werden dürfen.

Die folgenden Beispiele kommen zum größten Teil aus dem *Technical Report on C++ Performance*.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C++ und ROM

Wenn man etwas in C rommen darf, darf man es auch in C++ rommen:

```
static const int table[] = { 1, 2, 3 };        // table ist ROMbar
const char *pc1 = "Hello World";               // "Hello World" ist ROMbar
                                               // (aber pc1 nicht)
const char * const pc2 = "World";              // "World" is ROMbar (und
                                               // darf mit "Hello World“
                                               // geteilt werden);
                                               // pc2 ist auch ROMbar
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## C++ und ROM

Integrale, skalare Konstanten mit statischer Speicherdauer sind ROMbar, aber oftmals sind sie komplett wegoptimiert:

```
const unsigned bufSize = 128;          // i.d.R. wegoptimiert, es sei denn
                                       // die Adresse von bufSize
                                       // wird benutzt
```

Enums fordern keinen Speicher, und sie sind sicherer als #defines:

```
enum { bufSize = 128 };                // fast immer wegoptimiert
```

- Keine portable Art, die Größe eines enum-Werts zu bestimmen.
  - ➧ C++11 erlaubt Spezifikation des zugrundeliegenden Typs:

```
enum: unsigned short { bufSize = 128 };          // C++11
```

Solche Konstanten werden meistens Direktoperanden in Maschinenbefehlen.

- Falls nicht, dürfen sie auf jeden Fall in das ROM geschrieben werden.
- In diesen Fällen haben #defines keine Vorteile.
  - ➧ #defines respektieren keine Scopes.
  - ➧ #defines dürfen nicht private oder protected sein.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C++ und ROM

Für nicht-integrale, skalare Konstanten sind consts sicherer als #defines und möglicherweise effizienter:

```
#define pi 3.14159                  // ROMbar, aber leidet an
                                    // Makro-Nachteilen

const double pi = 3.14159;          // ROMbar, aber vermeidet
                                    // Makro-Nachteile
```

Gleitkommawerte können selten in Direktoperanden verwandelt werden:

- Beide obigen Formen sind ROMbar.
- Mit einem schlechten Compiler könnte das Makro mehrere Kopien von pi in einer Objektdatei stecken lassen.
  - ➡ Das sollte mit dem const nicht geschehen.
    - ◆ Es sollte höchstens eine Kopie in einer Objektdatei zulassen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## C++ und ROM

Objekte dürfen in das ROM geschrieben werden, wenn folgendes wahr ist:

- Sie sind const am Definierungsort deklariert.
- Sie enthalten keine mutable Datenelemente.
- Sie sind mit zur Kompilierzeit bekannten Werten initialisiert.
  - ➡ Solches Wissen könnte aus Flußanalyse usw. kommen.

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

const Point origin = { 0, 0 };          // origin ist ROMbar

struct Widget {                         // alle Widgets können bitweise
  int a;                                // initialisiert werden von einem in das
  const char *p;                        // ROM geschrieben Widget, das mit
  Widget(): a(7), p("xyzzy") { }        // { 7, "xyzzy" } initialisiert wurde
};

const Widget w;                         // w ist ROMbar (auch wenn es
                                        // ein Nicht-POD ist, die dynamische
                                        // Initialisierung erfordert)
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C++ und ROM

Kapselung der POD-Typen ist begrenzt, aber Wrapper können helfen:

```
class Widget {                          // Nicht-POD (hat private Daten)
  int x, y;
};
const Widget w = { 0, 0 };              // ungültig – w ist kein Aggregat

struct Widget {                         // POD (und ein Aggregat)
  int x, y;
};
const Widget w = { 0, 0 };              // i.d.R. wird gerommt

class WidgetWrapper {
  struct Widget { int x, y; }           // POD
  static const Widget w;                // ROMbar
};
const WidgetWrapper::Widget
  WidgetWrapper::w = { 0, 0 };          // i.d.R. wird gerommt
```

Details in Heritys 1998 *Embedded Systems Programming* Artikel.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## C++ und ROM

Einige Compiler-erzeugte Datenstrukturen lassen sich normalerweise in das ROM schreiben:

- Vtbls
- RTTI Tabellen und type_info Objekte
- Tabellen für Ausnahmebehandlung

Solche Objekte in das ROM zu schreiben ist allerdings unmöglich, wenn dynamisches Linken und Shared Libraries eingesetzt werden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C++ und ROM

Was ist nicht ROMbar? Objekte, die zur Laufzeit geändert werden dürfen.

- Objekte mit nicht-trivialen Konstruktoren oder Destruktoren.

```
class Widget {
public:
  Widget();                    // Widget Objekte nicht ROMbar
  ...
};
```

- Objekte mit mutable Datenelementen.

```
class Widget {
  mutable int lastValue;       // Widget Objekte nicht ROMbar
  ...
};
```

- Objekte, die nicht const definiert sind.

```
int x = 14;                    // x ist nicht const, also nicht ROMbar
std::string s = "xyzzy";       // s ist nicht const, also nicht ROMbar
```

➧ Selbstverständlich dürfen 14 und "xyzzy" gerommt werden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Zusammenfassung: C++ und ROM

- Die meisten Compiler/Linker lassen statisch initialisierte POD-Typen rommen.
  ➧ Aggressive Compiler können noch weiter gehen.
- Man kann ROMbare PODs dadurch kapseln, sie protected oder private in einen nicht-POD zu stellen.
- Vom Compiler erzeugte Datenstrukturen sind normalerweise ROMbar.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Überblick

Tag 2 (voraussichtlich):

- Modellierung von memory-mapped I/O
- Implementierung von Callbacks für C Interfaces
- Interessante Anwendungen von Templates:
  - ➧ Typsichere void*-basierte Container
  - ➧ Analyse von physikalischen Maßeinheiten zur Kompilierzeit
  - ➧ Die Spezifikation von endlichen Automaten (falls die Zeit reicht)
- Überlegungen für sicherheitskritische und Echtzeitsysteme
- Weiterführende Informationen (Englisch)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die Modellierung von memory-mapped I/O

Viele Systeme bilden I/O-Geräte auf feste Bereiche des Adressraums des Programms ab.

- Ein- und Ausgaberegister sind oft getrennt.
- Status-/Kontrollregister sind oft von Datenregistern getrennt.
  - ➧ Verschiedene Bits im Statusregister vermitteln Informationen wie z.B. ob das Gerät bereit ist oder ob Interrupts eingeschaltet sind.

Mit C++ ist es leicht, memory-mapped Geräte als Objekte mit natürlichen Interfaces darzustellen.

- Mit keinem Laufzeitaufwand.
  - ➧ Vorausgesetzt, dass Sie einen anständigen Compiler haben :-)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Die Modellierung von memory-mapped I/O

Memory-mapped Geräte könnten Sonderbehandlung benötigen, z.B

- Atomares Lesen/Schreiben könnte explizite Synchronisation erfordern.
- Einzelne Bits könnten mal schreibgeschützt sein, mal lesegeschützt.
- Um ein Bit zu löschen muss man es vielleicht auf *1* setzen.
- Ein Statusregister könnte mehr als ein Datenregister kontrollieren.
  ➧ Z.B. sind Bits 0-3 für ein Register, Bits 4-7 für ein anderes.

In Folgenden beschreibe ich einen MMIO-*Ansatz*, nicht eine Vorschrift.

- Der Ansatz schildert, wohin man die benötigte Sonderbehandlung der Geräte stellen soll.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Ein Kontrollregister modellieren

Nehmen wir an, dass wir ein 4-Byte Kontrollregister haben:

- Bit 0 entspricht Bereitschaft: 1 ⇒ bereit, 0 ⇒ nicht bereit.
- Bit 2 entspricht Interruptstatus: 1 ⇒ eingeschaltet, 0 ⇒ nicht.

Mit 4-Byte ints kann man das Register wie folgt darstellen:

```
enum { bit0 = 0x1, bit1 = 0x2, ... , bit31 = 0x80000000 };

class ControlReg {
public:
  bool ready() const                { return regValue & bit0; }
  bool interruptsEnabled() const    { return regValue & bit2; }
  void enableInterrupts()           { regValue |= bit2; }
  void disableInterrupts()          { regValue &= ~bit2; }

private:
  volatile unsigned regValue;          // Daten im Register könnten
};                                     // sich spontan ändern
```

Alle Funktionen sind inline, also sollten sie keinen Laufzeitsaufwand haben.

- Angenommen, dass sie wirklich geinlined werden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Ein Kontrollregister modellieren

Anmerkungen:

- In diesem Beispiel nehmen wir an, dass unsigned int die richtige Bitbreite und Ausrichtung für das Register hat:
  - ➡ Falls nicht, müssen Sie einen passenderen Datentyp auswählen.
  - ➡ Der Header <stdint.h> (mit z.B. uint32_t) kann hilfreich sein.
    - ◆ Standard in C99 und C++11.
- Wir nehmen auch an, dass das Register lesbar als auch schreibbar ist.
  - ➡ Falls es lesegeschützt ist, muss man den letzten geschriebenen Wert speichern.
    - ◆ Ein Beispiel kommt später.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Masken vs. Bitfelder

- Mein Code verwendet Bitmasken anstatt Bitfelder.
  - ➡ Compiler müssen die "offensichtliche" Bitfelder-Abbildung nicht nutzen:

```
class ControlReg {                                  // unzuverlässiger
public:                                             // Entwurf
  bool ready() const { return readyBit; }
  ...
private:
  volatile unsigned  readyBit              :1,      // Unzuverlässig!
                     /* frei */            :1,      // Compiler müssen
                     interruptEnabledBit   :1,      // readyBit nicht auf
                     /* frei */            :29;     // Bit 0 abbilden (usw.)
};
```

  - ➡ Auf einigen Plattformen könnten Bitfelder dennoch schneller sein.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Nebenbemerkung: new und Placement new

Ein *new Ausdruck* wie `T *p = new T` hat zwei Wirkungen:

1. Ruf eine operator new Funktion auf, um herauszufinden, wohin das T Objekt gestellt werden soll.
2. Ruf den passenden T Konstruktor auf.

**Wichtig:** der wesentliche Job von operator new ist nicht, Speicher zu allozieren, sondern zu vermitteln, *wohin ein Objekt gestellt werden soll*.

- Normalerweise erfolgt dynamische Speicherallokation.
- Manchmal weiß man, wohin ein Objekt gestellt werden soll:
  - ➧ Man hat eine MMIO-Adresse.
  - ➧ Man hat einen Puffer, der ein Objekt enthalten soll.
- Man kann operator new einen Zeiger auf die gewünschte Adresse übergeben, und sie gibt diesen Wert zurück:

```
void* operator new(std::size_t, void *ptrToMemory)
{ return ptrToMemory; }
```

- Diese Form von operator new heißt *placement new*.
  - ➧ Sie ist standardmässig und überall verfügbar.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Nebenbemerkung: new und Placement new

Weil ein Ausdruck wie `T *p = new T` zwei Funktionen aufruft, müssen wir zwei Parameterlisten übergeben können.

- Argumente an Konstruktor übergeben: `T *p = new T(Ctor args);`
- Argumente an operator new übergeben: `T *p = new (op new args) T;`
- Beides: `T *p = new (op new args) T(Ctor args);`
  - ➧ Man kann deshalb einen beliebigen Konstruktor aufrufen, wenn man placement new verwendet.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Ein Kontrollregister modellieren

Memory-mapped I/O Register haben festgelegte Adressen. Placement new ermöglicht, dass ein ControlReg Objekt die korrekte Adresse hat.

```
// ein ControlReg Objekt mit Adresse 0xFFFF0000 schaffen,
// und vereinbaren, dass pcr darauf hinweist

ControlReg * const pcr =
  new (reinterpret_cast<void*>(0xFFFF0000)) ControlReg;
```

- Erinnerung: durch placement new wird kein Speicher alloziert.

Über pcr kann man das Gerät manipulieren:

```
while (!pcr->ready()) ;                 // abwarten bis readyBit gesetzt ist
pcr->enableInterrupts();                // Gerät-Interrupts aktivieren
if (pcr->interruptsEnabled()) ...       // wenn Interrupts aktiviert sind...
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Ein Kontrollregister modellieren

Um die Zeiger-Syntax zu vermeiden, kann man den dereferenzierten Zeiger an eine Referenz binden:

```
ControlReg& cr =
  *new (reinterpret_cast<void*>(0xFFFF0000)) ControlReg;
while (!cr.ready()) ;                   // abwarten bis readyBit gesetzt ist
cr.enableInterrupts();                  // Gerät-Interrupts aktivieren
if (cr.interruptsEnabled()) ...         // wenn Interrupts aktiviert sind...
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Placement new vs. rohe Casts

Unser Einsatz von placement new ruft den Standardkonstruktor von ControlReg auf.

- Er ist implizit, inline und leer.
  - ➧ Er sollte wegoptimiert werden (d.h. keine Befehle erzeugen).
  - ➧ Wenn das nicht passiert, und das stört Sie, können Sie einen reinterpret_cast als Ersatz für placement new in Betracht ziehen:

```
ControlReg* const pcr =                                  // Zeiger-Version
  reinterpret_cast<ControlReg*>(0xFFFF0000);

ControlReg& cr =                                         // Referenz-Version
  *reinterpret_cast<ControlReg*>(0xFFFF0000);
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Placement new vs. rohe Casts

Placement new ist in der Regel einem rohen reinterpret_cast vorzuziehen:

- Es funktioniert richtig, wenn der Ctor des Geräts etwas zu tun hat.
- In diesem Fall zeigt ein roher Cast das falsche Verhalten.

Aber reinterpret_cast kann manchmal besser sein:

- Wenn placement new nicht total wegoptimiert wird (und das stört Sie).
- Wenn Sie die Adresse von einem I/O-Register rommen wollen *und*
  - ➧ der Compiler rommt das Ergebnis eines reinterpret_casts *und*
  - ➧ er rommt das Ergebnis von placement new nicht.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Objektplatzierung via Compilererweiterungen

Für einige Compiler/Linker gibt es eine weitere Möglichkeit:

- Verwendung von Compilererweiterungen, um ein Objekt auf eine spezifische Adresse zu platzieren.

Beispiele:

- Altium Tasking Compiler bieten diese Art von Syntax an:

```
ControlReg cr __at(0xFFFF0000);
```

- Der Wind River Diab Compiler bietet das an:

```
#pragma section MMIO address=0xFFFF0000
#pragma use_section MMIO cr
ControlReg cr;
```

Solche Erweiterungen könnten Beschränkungen auferlegen:

- z.B. die Objekte müssen von POD-Typen sein.

Vorteile:

- Kein Bedarf, auf MMIO-Objekte über einen Zeiger zuzugreifen.
- Kein Bedarf, Speicher für einen Zeiger auf jedes MMIO-Objekt zu reservieren.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Objektplatzierung via Linkerbefehle

Linker könnten das Spezifizieren von Objektadressen unterstützen:

- C++-Quellcode ist "normal."
  - ➡ Er enthält keine Information zur Objektplatzierung.

```
ControlReg cr;                    // in C++-Quelldatei
```

- Linkerskripte bilden C++-Objekte auf Speicheradressen ab, oft durch:
  - ➡ Das Abbilden von Objekten auf Sektionen.
  - ➡ Das Abbilden von Sektionen auf Adressbereiche.

Das Ergebnis ist portablerer Code.

- Plattformspezifische Adressen erscheinen nur in Linkerskripten.
- "Hardwareingenieure üben ihre Schreckensherrschaft auf jemand anders aus."


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Modell eines Ausgabegerätes, 1. Version

Man kann ein Ausgabegerät mit int-Größe als eine Verbindung zwischen einem int und dem entsprechenden ControlReg Objekt darstellen:

```
class OutputDevice1 {
public:
  OutputDevice1(unsigned controlAddr,
                unsigned dataAddr);          // Def. auf nächster Folie
  ControlReg& control() { return *pcr; }     // Zugriff auf ControlReg
  void write(unsigned value)                 // Daten auf das Gerät
  { *pd = value; }                           // schreiben
private:
  ControlReg* const pcr;                     // ptr auf ControlReg
  volatile unsigned * const pd;              // ptr auf Daten
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Modell eines Ausgabegerätes, 1. Version

Der Konstruktor lässt pcr und pd auf die richtige Adressen zeigen:

```
inline
OutputDevice1::OutputDevice1(unsigned controlAddr, unsigned dataAddr)
:  pcr(new (reinterpret_cast<void*>(controlAddr)) ControlReg),
   pd(new (reinterpret_cast<void*>(dataAddr)) unsigned)
{}
```

Clientverwendung sieht so aus:

```
OutputDevice1 od( 0xFFFF0000,          // Adr. des ControlRegs
                  0xFFFF0004);         // Adr. des DatenRegs

unsigned x;

...

while (!od.control().ready()) ;        // abwarten, bis readyBit gesetzt ist

od.write(x);                           // x auf od schreiben

...
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Modell eines Ausgabegerätes, 2. Version

MMIO Adressen sind Konstanten zur Kompilierzeit, also sollte es nicht nötig sein, sie als Datenelemente wie pcr und pd zu speichern:

Ein Template mit Nicht-Typ Parametern ermöglicht so ein Entwurf:

```
template<unsigned controlAddr, unsigned dataAddr>
class OutputDevice2 {
public:
                                        // kein Ctor mehr
  static ControlReg& control()
  { return *reinterpret_cast<ControlReg*>(controlAddr); }
  static void write(unsigned value)
  { *reinterpret_cast<volatile unsigned*>(dataAddr) = value; }
                                        // keine Datenelemente mehr
};
```

OutputDevice2 benutzt statische Member-Funktionen, um keinen this-Zeiger übergeben zu müssen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Modell eines Ausgabegerätes, 2. Version

Das Beispiel nimmt an, dass die Ctor und Dtor von ControlReg nichts tun.

- Sonst braucht OutputDevice2 einen Ctor/Dtor, die sie aufrufen (z.B. via placement new).

```
template<unsigned controlAddr, unsigned dataAddr>
class OutputDevice2 {
public:
  OutPutDevice2()
  { new reinterpret_cast<ControlReg*>(controlAddr) ControlReg; }
  …
};
```

- Solche Initialisierung/Aufräumen muss nur einmal stattfinden!
  - ➡ Problematisch, wenn es mehrere OutputDevice2-Objekte für ein einziges Gerät gibt.
    - Singleton einsetzen, um mehrfache Instanzen zu verhindern?
    - Statische alreadyInitialized/alreadyCleanedup Flags nutzen?


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Modell eines Ausgabegerätes, 2. Version

Client-Code sieht fast genauso aus wie zuvor:

```
OutputDevice2<0xFFFF0000, 0xFFFF0004> od;
unsigned x;
...
while (!od.control().ready()) ;          // abwarten, bis readyBit gesetzt ist
od.write(x);                             // x auf od schreiben
...
```

Vorteile dieses Ansatzes:

- OutputDevice2–Objekte sind kleiner als OutputDevice1–Objekte.
- OutputDevice2–Code könnte auch kleiner/schneller als OutputDevice1–Code sein.
  - ➧ Kein Bedarf, indirekt über den this –Zeiger zuzugreifen.

Danke an Siegward Jäkel für die Idee dieses Entwurfs.



## Modell eines Ausgabegerätes, 3. Version

Wenn, wie in diesem Beispiel, die Kontroll- und Datenregister in benachbartem Speicher sind, kann man einen dritten Entwurf verwenden:

- Objekte *direkt an den MMIO Adressen* erstellen:

```
class OutputDevice3 {
public:
  ControlReg& control()       { return cr; }
  void write(unsigned value) { data = value; }
private:
  OutputDevice3(const OutputDevice3&);          // Kopieren verhindern
  ControlReg cr;
  volatile unsigned data;
};
```

- Clients benutzen placement new (oder rohen reinterpret_cast) direkt:

```
// OutputDevice3-Objekt an der Adresse 0xFFFF0000 erstellen
OutputDevice3& od =
  * new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3;
```



Scott Meyers, Fachberater für Softwareentwicklung
http://www.aristeia.com/
© 2011 Scott Meyers, alle Rechte vorbehalten.

## Direkte Modellierung von Hardware

Es gibt also zwei Arten von Klassen, die memory-mapped I/O modellieren.

Eine Art modelliert Hardware *direkt*.

- Objekte solcher Klassen werden von Clients an spezifischen Adressen erstellt.
  - ➧ Via placement new oder reinterpret_cast.
- Sie enthalten nur nicht-statische Daten, die auf MMIO-Register abgebildet werden:

```
class OutputDevice3 {                  // Klasse, die entworfen ist,
public:                                // nur auf MMIO-Adressen
  ...                                  // erstellt zu werden
private:
  OutputDevice3(const OutputDevice3&);
  ControlReg cr;                       // Datenelemente bilden sich direkt
  volatile unsigned data;              // auf Geräteregister ab
};
```

  - ➧ Statische Daten sind okay.
- Sie haben niemals virtuelle Funktionen.
  - ➧ Virtuelle Funktionen führen zu einem vptr *irgendwo* in jedem Objekt.



## Indirekte Modellierung von Hardware

Die andere Art modelliert Hardware *indirekt*.

- Objekte solcher Klassen werden *nicht* an spezifischen Adressen erstellt.
  - ➧ Clients übergeben MMIO-Adressen als Template- oder Konstruktor-Argumente.
- Sie dürfen "extra" Datenelemente enthalten.
  - ➧ d.h. die keinem MMIO Geräteregister entsprechen.

```
class OutputDevice1 {
  ...
private:
  ControlReg* const pcr;               // Datenelemente, die sich auf keine
  volatile unsigned * const pd;        // MMIO-Geräteregister abbilden

  unsigned lastValueWritten;           // nützlich für lesegeschützte
};                                     // Register
```



Scott Meyers, Fachberater für Softwareentwicklung
http://www.aristeia.com/
© 2011 Scott Meyers, alle Rechte vorbehalten.

## Indirekte Modellierung von Hardware

- Sie dürfen virtuelle Funktionen haben:

```
class DeviceBase {
public:
  virtual void reset() = 0;
  …
};

class OutputDevice1: public DeviceBase {
public:
  virtual void reset();
  …
};

template<unsigned controlAddr, unsigned dataAddr>
class OutputDevice2: public DeviceBase {
public:
  virtual void reset();
  …
};

…                                   // fortgesetzt auf der nächsten Folie…
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Indirekte Modellierung von Hardware

```
OutputDevice1 od1a(0xFFFF0000, 0xFFFF0004);
OutputDevice1 od1b(0xFFFF0010, 0xFFFF0014);
…
OutputDevice2<0xEEEE0000, 0xEEEE0010> od2a;
OutputDevice2<0xEEEE0020, 0xEEEE0040> od2b;
…
DeviceBase* registers[] = { &od1a, &od1b, …, &od2a, &od2b, … };
const std::size_t numRegisters = sizeof(registers)/sizeof(registers[0]);
…
for (std::size_t i =0; i < numRegisters; ++i)        // reset alle Register
  registers[i]->reset();                             // im System
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Indirekte Modellierung von Hardware

Indirekte Modellierung aufwändiger als direkte Modellierung:

- Speicher für "extra" Datenelemente (falls vorhanden)
- Indirektion zwischen Objekt und Register (falls benötigt)

Sie ist auch flexibler:

- Man darf andere Datenelemente zufügen.
- Man darf virtuelle Funktionen deklarieren.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Mögliche Nutzungsfehler verhindern

Klassen, die Hardware direkt modellieren, können leicht falsch eingesetzt werden, z.B:

- Clients könnten sie auf nicht-MMIO Adressen instantiieren.
- Clients, die placement `new` verwenden, könnten denken, dass sie `delete` verwenden sollen.
  ➧ Stimmt nicht, aber vielleicht müssen sie den Destruktor aufrufen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Mögliche Nutzungsfehler verhindern

Um solche Fehler zu verhindern, kann ein privater Destruktor hilfreich sein:

```
class OutputDevice3 {                   // Klasse, die Hardware direkt
  ...                                   // modelliert; verhindert einige
private:                                // Nutzungsfehler
  ~OutputDevice3() {}
  ControlReg cr;
  volatile unsigned data;
};
OutputDevice3 d;                        // Fehler! impliziter Aufruf des
                                        // Dtors. (Wir wollen vorbeugen,
                                        // dass MMIO-Objekte auf
                                        // den Stack gestellt werden.)
OutputDevice3* pd =
  new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3;      // OK
...
delete pd;                              // Fehler! noch ein impliziter
                                        // Aufruf des Dtors
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Mögliche Nutzungsfehler verhindern

Aber wenn der Destruktor Arbeit verrichten muss, muss er manuell aufgerufen werden.

- Das impliziert, dass der Destruktor public sein muss:

```
class OutputDevice3 {
public:
  ~OutputDevice3();
  ...
};
OutputDevice3* pd =
  new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3;
...
pd->~OutputDevice3();
```

Aber wir haben eben gesehen, dass public Destruktoren zu Nutzungsfehlern führen können....


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Mögliche Nutzungsfehler verhindern

Man kann beides haben, wenn der Destruktor indirekt aufgerufen wird:

```
class OutputDevice3 {
public:
  void destroy() { this->~OutputDevice3(); }          // ohne "this->" wird es
  ...                                                 // unzutreffend geparst
private:
  ~OutputDevice3() { ... }
  ...
};
OutputDevice3 d;                                      // bleibt ein Fehler
OutputDevice3* pd =                                                  // bleibt
  new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3;           // OK
...
delete pd;                                            // bleibt ein Fehler
pd->~OutputDevice3();                                 // Fehler!
pd->destroy();                                        // OK
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Mögliche Nutzungsfehler verhindern

Clients können MMIO-Geräte immer noch auf unpassende Adressen setzen, aber wir können auch das verhindern. Ein Ansatz:

```
class InDevCtrlRegAddr { ... };          // Klassen, die gültige MMIO-
class OutDevCtrlRegAddr { ... };         // Adressen darstellen
...
const OutDevCtrlRegAddr ODCRA1(0xFFFF0000);     // ein Objekt für jede
...                                             // MMIO-Adresse
class OutputDevice {
public:
  static void* operator new( std::size_t,                // op. new, der nur
                             OutDevCtrlRegAddr);         // MMIO-Objekte nimmt
  ...
};
OutputDevice& od = *new (ODCRA1) OutputDevice(params);
```

Hier muss man kein reinterpret_cast benutzen, wenn man Objekte erzeugt.

- Es muss aber möglich sein, void* aus einem OutDevCtrlRegAddr zu erhalten.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Mögliche Nutzungsfehler verhindern

Die Details dieses Ansatzes bleiben Ihnen überlassen, aber:

- Ein grundsätzliches Entwurfsziel ist, dass *Verletzungen des Entwurfs nie kompilieren sollten*.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Verallgemeinerung durch Templates

Geräte unterscheiden sich oft in vielerlei Hinsicht:

- Bitbreiten der Register.
- Bits, die Bereitschafts- und Interruptstatus entsprechen, usw.

Mit Templates kann man diese Variationen leicht ausdrücken:

```
template<typename RegType,
         RegType ReadyBitMask,
         RegType InterruptBitMask>
class ControlReg {
public:
  bool ready() const                { return regValue & ReadyBitMask; }
  bool interruptsEnabled() const { return regValue & InterruptBitMask; }
  void enableInterrupts()           { regValue |= InterruptBitMask; }
  void disableInterrupts()          { regValue &= ~InterruptBitMask; }

private:
   volatile RegType regValue;
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Verallgemeinerung durch Templates

```
// CRui03 ist ein ControlRegister mit Bitbreite von unsigned int.
// Bit 0 ist das Ready-Bit und Bit 3 ist das Interrupt-Bit.
typedef ControlReg<unsigned int, bit0, bit3> CRui03;
CRui03& cr1 = * new (reinterpret_cast<void*>(0xFFFF0000)) CRui03;
...                                                    // cr1 benutzen

// CRuc15 ist ein ControlRegister mit Bitbreite von unsigned char.
// Bit 1 ist das Ready-Bit und Bit 5 ist das Interrupt-Bit.
typedef ControlReg<unsigned char, bit1, bit5> CRuc15;
CRuc15& cr2 = * new (reinterpret_cast<void*>(0xFFFF0010)) CRuc15;
...                                                    // cr2 benutzen
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Zusammenfassung: Die Modellierung von memory-mapped I/O

C++ Tools, die Sie wahrscheinlich einsetzen wollen:

- Klassen
- Klassentemplates (mit Typ- und auch nicht-Typ Parametern)
- Inline-Funktionen
- Placement new und reinterpret_cast
- const Zeiger
- volatile Speicher
- Referenzen
- Private Member-Funktionen, z.B. Kopierkonstruktor, Destruktor.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Überblick

Tag 2 (voraussichtlich):

- Modellierung von memory-mapped I/O
- Implementierung von Callbacks für C Interfaces
- Interessante Anwendungen von Templates:
  - ➧ Typsichere void*-basierte Container
  - ➧ Analyse von physikalischen Maßeinheiten zur Kompilierzeit
  - ➧ Die Spezifikation von endlichen Automaten (falls die Zeit reicht)
- Überlegungen für sicherheitskritische und Echtzeitsysteme
- Weiterführende Informationen (Englisch)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die Implementierung von Callbacks aus C

APIs unterstützen oftmals C Callback-Funktionen, z.B.:

- Hardwareinterrupts ⇒ Aufrufe von ISRs in C.
- Signale vom Betriebssystem ⇒ Aufrufe von Signal-Handler in C.
- Anwendungs-Events ⇒ Aufrufe von Event-Handler in C.

Ziele:

- Die Callbacks in C++ schreiben.
- Volle Flexibilität behalten:
  - ➧ Zugriff auf alle C++ Sprachfeatures.
  - ➧ Callbacks dynamisch erstellen/konfigurieren/installieren/ersetzen können.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Die Implementierung von Callbacks aus C

Details von Callback-APIs sind unterschiedlich:

- Parameters der Callback-Funktion, z.B.:

```
void callbackFcn();                                  // keine params
void callbackFcn(int eventID);                       // nur eventID
void callbackFcn(int eventID, void *pEventDaten);    // eventID + beliebige
                                                     // userdefinierte Daten
```

  - ➧ Andere Parameterlisten (Typen, Anzahl der Parameters) sind möglich.
- Rückgabetypen müssen nicht immer void sein.
- Beschränkungen auf Callback-Verhalten:
  - ➧ z.B. ISRs und Signal-Handler müssen schnell sein, sicher beim asynchronen Aufruf, usw.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## ISRs als Callback-Beispiele

Das Beispiel ist ISRs, aber der Schwerpunkt sind Entwurfsentscheidungen:

- Wirklicher ISR-Code ist komplizierter:
  - ➧ Benötigt evtl. spezielle Aufrufskonventionen.
  - ➧ Verhalten oft beschränkt:
    - ◆ Während Ausführung müssen andere Interrupts vielleicht ausgeschaltet werden.
    - ◆ Dürfen nur auf atomare volatile Daten sicher zugreifen.
    - ◆ Könnte eine harte Begrenzung der Ausführungszeit haben.
- Wir ignorieren solche Details.
  - ➧ Sie beinflussen die grundsätzlichen Entwurfsansätze nicht.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beispiel für ein C Callback API

Nehmen wir an, dass diese ISR API in C implementiert ist:

```
typedef void (*ISR_t)(int);              // param = ID des Interrupts
void setISR(int interruptID, ISR_t isr);
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## C-ähnliche Funktionszeiger

Man darf zwei Arten von C++ Funktionszeigern an setISR übergeben:

- Nicht-Member-Funktionen, z.B. Funktionen in globalem oder Namensraum-Scope.
- Statische Member-Funktionen.

Grund: Sie haben keine this-Zeiger.

- Zeiger auf nicht-statische Member-Funktionen fordern einen this Zeiger.
  - ➧ Ihr Speicherlayout ist anders als das von "normalen" Funktionszeigern.

Statische Member-Funktionen sind vorzuziehen:

- Keine Verschmutzung des Namensraums: die Namen sind lokal in der Klasse.
- Zugriffskontrolle: die Funktionen können protected oder private sein.
- Zugriffsfähigkeiten: sie haben Zugriff auf protected und private (statische) Klassen-Elemente.

Unser erstes Ziel ist, statische Member-Funktionen als Callbacks zu benutzen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C vs. C++ Linkage

*Linkage* kann problematisch sein:

- C Funktionen haben C Linkage.
- C++ Funktionen haben C Linkage, nur wenn sie als extern "C" deklariert werden.

```
void f1(int);                   // Nicht-Member-Fktn:  standardmäßige
                                // C++ Linkage

extern "C" void f2(int);        // Nicht-Member-Fktn:  explizit C Linkage

class Widget {
public:
  static void smf(int);         // Member-Fktn:  immer C++ Linkage
};
```

Für C++ Kompilierung brauchen ISR_t und setISR C Linkage, also:

```
extern "C" {                    // nutzen C Aufrufkonventionen
  typedef void (*ISR_t)(int);
  void setISR(int interruptID, ISR_t isr);
}
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## C vs. C++ Linkage

Compiler *dürfen* C und C++ Linkage unterschiedlich behandeln:

```
setISR(0, &f1);             // vielleicht kompiliert/linkt/läuft es – oder nicht
setISR(1, &Widget::smf);    // das gleiche

setISR(2, &f2);             // typischerweise kompiliert/linkt/läuft es
```

In der Regel sind nur Nicht-Member extern "C" Funktionen gültig als C Callbacks.

- Selbst dann nur für C und C++ Code, der binär kompatibel ist.
  - ➡ Es gibt kein Standard-Binärinterface für C oder C++ Linkage.
    - ◆ d.h. kein ABI ("Application Binary Interface").


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Statische Member-Funktionen als Callbacks

Die Nicht-Member kann eine statische Member-Funktion inline aufrufen:

```
class InterruptMgr {
public:
   …
   static void isr(int interruptID)       // “wirkliche” ISR
   { Code, der den Interrupt behandelt; }
};
extern "C" {
   void isrHelper(int interruptID)      // Funktion, die zur C API übergeben wird
   { InterruptMgr::isr(interruptID); }  // inline Aufruf auf wirkliche ISR
}
setISR(0, &isrHelper);                  // Callback installieren
```

Auf einigen Plattformen darf die Nicht-Member-Funktion wegfallen:

- Auf solchen Plattformen sind C und C++ Linkage gleich.

```
setISR(0, &InterruptMgr::isr);      // funktioniert auf vielen Plattformen
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Statische Member-Funktionen als Callbacks

Normalerweise ruft die statische Member-Funktion eine andere Funktion auf, die den Interrupt schlussendlich behandelt, also:

```
class InterruptMgr {
public:
   …
   static void isrDispatcher(int interruptID)          // neuer Name
   { ruft Interrupt behandelnde Funktion auf; }
};
extern "C" {
   void isrHelper(int interruptID)
   { InterruptMgr::isrDispatcher(interruptID); }
}
setISR(0, &isrHelper);                                 // wie vorher
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Propagierung von Exceptions in C verhindern

C++ Exceptions dürfen in C nicht propagiert werden.

- Layout eines Stack-Frames für C könnte anders als für C++ sein!

Wenn Callbacks werfen können, muss man die Weitergabe verhindern, z.B.:

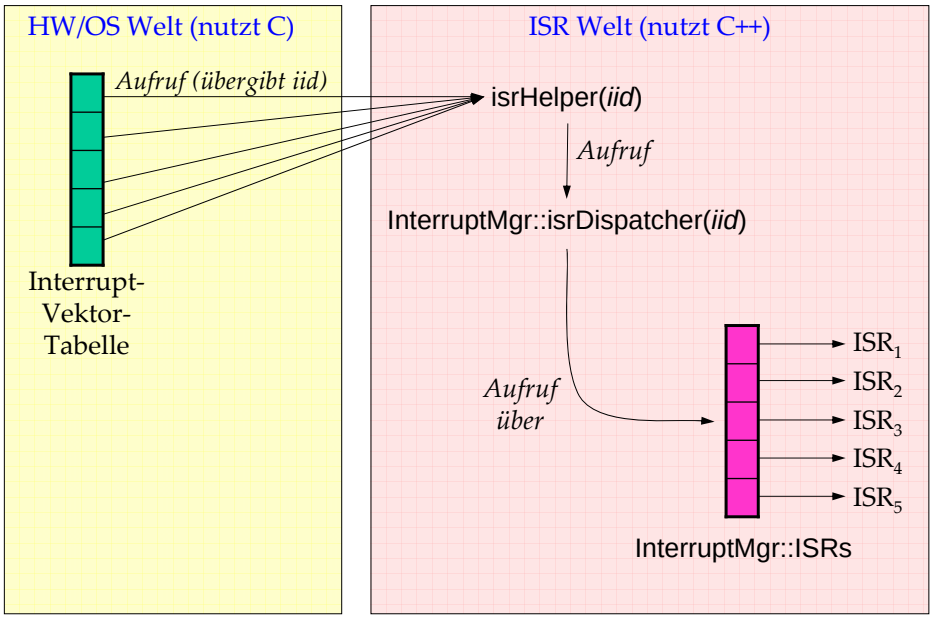
```
extern "C" {
  void isrHelper(int interruptID)   // Funktion, die zu C API übergeben wird
  {
    try {
      InterruptMgr::isrDispatcher(interruptID);
    }
    catch (…) {
      errno setzen, Exception loggen, irgendetwas….
    }
  }
}
```

Zur Laufzeit kann ein try-Block etwas aufwändig sein:

- Es könnte besser sein, Exceptions in Callbacks einfach zu verbieten.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die grundsätzliche Strategie für Callback-Behandlung





Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Die grundsätzliche Strategie für Callback-Behandlung

InterruptMgr funktioniert wie folgt:

```
class InterruptMgr {
public:
  typedef ??? ISRType;                          // Details kommen gleich
  …
  static void registerISR(int interruptID, ISRType isr)
  { ISRs[interruptID] = isr; }
  static void isrDispatcher(int interruptID)
  { ruf ISRs[interruptID](interruptID) auf; }    // Details kommen gleich
private:
  static ISRType ISRs[NUM_INTERRUPTS];          // dekl. Arr. von echten ISRs
};
InterruptMgr::ISRType                           // definieren Arr. von echten
  InterruptMgr::ISRs[NUM_INTERRUPTS];           // ISRs
```

Das ISRs-Array spiegelt die Interrupt-Vektor-Tabelle des Systems.

- Wir können das Array in C++ *irgendetwas* enthalten lassen:
  - ➧ Es könnte Objekte, Objekt-Zeiger, Mem.-Fktn.-Zeiger, usw. enthalten.
  - ➧ Wir sind jetzt in der Welt von C++.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Callbacks durch virtuelle Funktionen

Schreiben Sie eine Basisklasse für Objekte, die Interrupts behandeln:

```
class ISRBase {                                 // Klassen, die ISRs umsetzen,
public:                                         // vererben von dieser Klasse
  …
  virtual void isr(int interruptID) = 0;        // oder evtl. operator()(int)
};
```

InterruptMgr könnte dann so aussehen:

```
class InterruptMgr {
public:
  typedef ISRBase* ISRType;
  …
  static void registerISR(int interruptID, ISRType isr) { … }
  static void isrDispatcher(int interruptID)
  {
    ISRs[interruptID]->isr(interruptID);        // virtueller Aufruf von ISR
   }
private:
  static ISRType ISRs[NUM_INTERRUPTS];          // Array von Zeigern auf
};                                              // Objekte mit ISRs
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Callbacks durch virtuelle Funktionen

Abgeleitete Klassen bieten die echten ISRs:

```
class TimerISR: public ISRBase {
public:
   …
  virtual void isr(int interruptID);
};
```

```
class KeyboardISR: public ISRBase {
public:
   …
  virtual void isr(int interruptID);
};
```

Objekte dieser Typen werden dann erstellt und registriert:

```
TimerISR t;
KeyboardISR k;

InterruptMgr::registerISR(TIMER_INT_NUM, &t);

InterruptMgr::registerISR(KEYBOARD_INT_NUM, &k);
```

Ergebnis:

- Interrupt Nummer TIMER_INT_NUM wird durch t.isr behandelt.
- Interrupt Nummer KEYBOARD_INT_NUM wird durch k.isr behandelt.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Callbacks durch virtuelle Funktionen

Überlegen Sie, was im Fall eines Timer-Interrupts passiert:

1. C API ruft Nicht-Member-Funktion isrHelper auf.
2. isrHelper ruft statische Member-Funktion InterruptMgr::isrDispatcher auf.
3. InterruptMgr::isrDispatcher ruft Member-Funktion ISRs[TIMER_INT_NUM]->isr auf.
   - Dieser virtuelle Aufruf verweist auf t.isr (d.h. TimerISR::isr auf t).

InterruptMgr::isrDispatcher ist inline, also erwarten wir zur Laufzeit:

1. C API ruft isrHelper auf.
2. isrHelper ruft t.isr via vtbl auf.

Es ist ähnlich, wenn ein Keyboard-Interrupt stattfindet:

1. C API ruft isrHelper auf.
2. isrHelper ruft k.isr via vtbl auf.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beurteilung: Einsatz von virtuellen Funktionen

Vorteile:

- Klare, saubere "objekt-orientierte" Lösung.
- Heap-Objekte sind erlaubt, sind aber nicht erforderlich.
  - ➡ t und k könnten global, in einem Namensraum, oder statisch mit Datei-Scope sein.

Nachteile:

- Basisklasse und virtuelle Funktionen müssen eingeführt werden.
  - ➡ Virtuelle Funktionen ⇒ vtbl.
  - ➡ Basisklasse könnte existieren, nur um Callbacks zu unterstützen.
    - ◆ Viele kleine "Interface-Klassen" (und ihre Dateien) könnten entstehen.
- Echte ISRs müssen nicht-statische Member-Funktionen sein.
  - ➡ Auch wenn statische Member-Funktionen oder nicht-Member-Funktionen genügen würden.
    - ◆ Die nicht-statischen Member-Funktionen dürfen sie ja aufrufen.
- Echte ISRs müssen gleiche Signaturen haben (einschliesslich constheit).
  - ➡ Bis auf kovariante Rückgabetypen….


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Callbacks via std::tr1::function

TR1 ist ein Bericht, der neue Bibliotheksfunktionalität für C++11 spezifiziert:

- Bald sehen wir ihn näher an.
- Alle Features in TR1 sind im Namensraum std::tr1.

Die TR1 Funktionalität, die wir hier verwenden, ist:

- tr1::function: verallgemeinerter Funktionszeiger; enthält alles Aufrufbare.
- tr1::bind: schafft Funktionsobjekte, die etwas Aufrufbares und einige Parameter-Werte enthalten.
  - ➡ Kann mehr leisten, aber für unsere Zwecke reicht das aus.
  - ➡ Resultat oft in einem tr1::function Objekt gespeichert.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Funktiontypen und std::tr1::function

Der Typ einer Funktion ist ihre Deklaration ohne Namen:

```
void f1(int x);                    // Typ ist void(int)
double f2(int x, std::string& s);  // Typ ist double(int, std::string&)
```

Das Template-Parameter von std::function ist der Zieltyp ihrer Funktion:

```
std::tr1::function<void(int)> func;  // func kann alles Aufrufbare enthalten,
                                     // das mit Typ void(int) kompatibel ist
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Callbacks via std::tr1::function

Änderungen im InterruptMgr sind klein:

```
class InterruptMgr {
public:
  typedef std::tr1::function<void(int)> ISRType;
  …
  static void registerISR(int interruptID, ISRType isr) { … }
  static void isrDispatcher(int interruptID)
  {
    ISRs[interruptID](interruptID);          // Funktionsaufruf-Syntax
  }                                          // verwendet
private:
  static ISRType ISRs[NUM_INTERRUPTS];  // Array von tr1::function
};                                      // Objekten
```

Ein ISRType Objekt:

- enthält *alles Aufrufbare, dem ein* int *übergeben werden darf und das irgendetwas zurückgibt*.
  - ➧ Es ist ein verallgemeinerter Funktionszeiger.
- wird mit Funktionssyntax aufgerufen.
  - ➧ z.B. innerhalb InterruptMgr::isrDispatcher.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Callbacks via std::tr1::function

TimerISR und KeyboardISR sind unverändert, außer:

- Sie haben keine Basisklasse.
  ➧ ISRBase ist nicht mehr nötig.
- Die isr Funktionen müssen nicht virtuell sein.
  ➧ Ihre Signaturen (z.B. constheit) müssen auch nicht gleich sein.

```
class TimerISR {                    // keine Basisklasse
public:
  …
  void isr(int interruptID);        // nicht-virtuelle Funktion (nicht-const)
};
class KeyboardISR {                 // keine Basisklasse
public:
  …
  void isr(int interruptID) const;  // nicht-virtuelle Funktion (const)
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Callbacks via std::tr1::function

Um eine ISR mit InterruptMgr zu registrieren, erzeugen wir ein Funktionsobjekt, das enthält:

- Die ISR Member-Funktion.
- Das Objekt, auf das die Funktion aufgerufen werden soll.

```
TimerISR t;                                          // wie vorher
KeyboardISR k;

// im Fall eines Timer-Interrupts, ruf t.isr auf
InterruptMgr::registerISR( TIMER_INT_NUM,
                           std::tr1::bind(&TimerISR::isr, &t, _1));

// im Fall eines Keyboard-Interrupts, ruf k.isr auf
InterruptMgr::registerISR(KEYBOARD_INT_NUM,
                           std::tr1::bind(&KeyboardISR::isr, &k, _1));
```

- Details für bind kommen gleich.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Callbacks via std::tr1::function

Überlegen Sie was im Fall eines Timer-Interrupts passiert:

1. Wie zuvor ruft C API isrHelper auf.
2. Wie zuvor ruft isrHelper InterruptMgr::isrDispatcher auf.
3. InterruptMgr::isrDispatcher ruft die vom tr1::function Objekt gehaltene Member-Funktion ISRs[TIMER_INT_NUM] auf.
   - Dieser Aufruf verweist auf t.isr durch Member-Funktion-Zeiger.

InterruptMgr::isrDispatcher ist noch inline, also erwarten wir zur Laufzeit:

1. C API ruft isrHelper auf.
2. isrHelper ruft t.isr via Member-Funktion-Zeiger auf.

Es ist ähnlich, wenn ein Keyboard-Interrupt stattfindet:

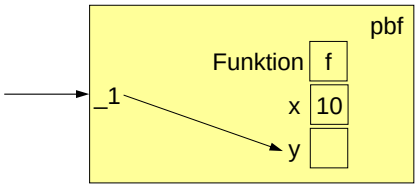
1. C API ruft isrHelper auf.
2. isrHelper ruft k.isr via Member-Funktion-Zeiger auf.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Details über std::tr1::bind

```
void f(int x, int y);                          // Funktion, die letztlich aufgerufen wird

std::tr1::function<void(int)> pbf =            // pbf = "partiell gebundene f";
   std::tr1::bind(f, 10, _1);                  //   x param von f auf 10 gesetzt
```




```
…

pbf(20);                                       // gleich wie f(10, 20)
```

Merke: pbf wird wie eine Funktion aufgerufen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# Details über std::tr1::bind

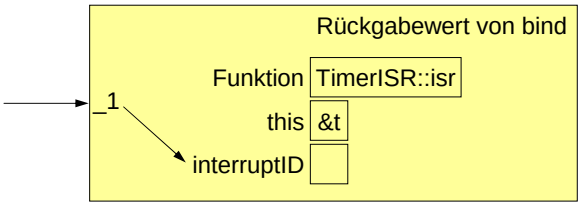
Wenn eine nicht-statische Member-Funktion gebunden wird, ist *this-Objekt Parameter Nr. 1:

```
class TimerISR {                        // wie vorher
public:
  …
  void isr(int interruptID);            // std::tr1::bind sieht zwei Param:
};                                      // *this ist Nr. 1, interruptID ist Nr. 2
```

Deswegen gibt

```
std::tr1::bind(&TimerISR::isr, &t, _1)
```

folgendes aus:



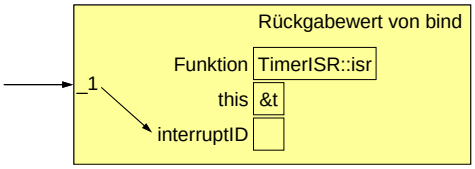


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.

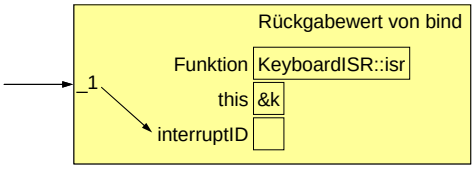

# Callbacks via std::tr1::function

Also:

```
TimerISR t;                                          // wie früher
KeyboardISR k;
InterruptMgr::registerISR( TIMER_INT_NUM,
                           std::tr1::bind(&TimerISR::isr, &t, _1));
```




```
InterruptMgr::registerISR( KEYBOARD_INT_NUM,
                           std::tr1::bind(&KeyboardISR::isr, &k, _1));
```





Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beurteilung: Einsatz von std::tr1::function

Vorteile:

- Kein Bedarf an benutzerdefinierten Basisklassen oder virtuellen Funktionen.
- Callbacks dürfen sein:
  - ➡ Funktionsobjekte:
    - ◆ z.B. gebundene nicht-statische Member-Funktionen, die von tr1::bind erstellt wurden.
  - ➡ Statische Member-Funktionen
  - ➡ Nicht-Member-Funktionen
- Signaturen der Callbacks müssen mit einer Zielsignatur lediglich kompatibel sein.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Beurteilung: Einsatz von std::tr1::function

Nachteile:

- TR1 Implementierungen noch nicht überall verfügbar.
  - ➡ Einige Compiler bieten nichts aus TR1 an.
    - ◆ Open-source und kommerzielle Versionen von bind und function stehen aber zur Verfügung.
  - ➡ Viele Entwickler sind mit den Bestandteilen von TR1 nicht vertraut.
- tr1::function Objekte bringen Ressourcenaufwände mit sich:
  - ➡ Sie könnten Heap-Speicher allozieren.
  - ➡ Einige Implementierungen verwenden virtuelle Funktionen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Zusammenfassung: Die Implementierung von Callbacks aus C

- Die Callbacks dürfen nicht nicht-statische Member-Funktionen sein.
  - Einige Plattformen gestatten statische Member-Funktionen.
  - Einige unterstützen nur nicht-Member-Funktionen, die extern "C" deklariert sind.
- Zwei grundsätzliche Callbackansätze:
  - Virtuelle Funktionen.
  - tr1::function Objekte.
- Ansätze unterscheiden sich in mehreren Eigenschaften:
  - Ob man Basisklassen und virtuelle Funktionen deklarieren muss.
  - Ob nicht-Member-Funktionen unmittelbar unterstützt sind.
  - Ob Callbacks mit einer bestimmten Signatur genau übereinstimmen müssen.
  - Ob nicht-normgerechte Bibliotheken (z.B. TR1) benutzt werden.
  - Ob Heap-Speicher und/oder vtbls verwendet werden.
  - Aufrufgeschwindigkeit.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Was ist TR1?

- "Technical Report 1" vom Bibliotheksausschuss des Standardkomitees.
- Basis für neue Bibliotheksfunktionalität in C++11.
- TR1 Funktionalität ist im Namensraum std::tr1.
- TR1-abgeleitete Funktionalität in C++11 ist in std.
  - Diese Funktionalität nicht gleich wie in TR1.
    - Verwendet neue C++11 Spracheigenschaften.
    - Die APIs wurden aus der Erfahrung mit TR1 angepasst.
  - Interfaces meistens rückwärtskompatibel.
    - C++11 bietet "verbesserte" TR1 Funktionalität.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## TR1 Zusammenfassung

| Neue Funktionalität | Kurzfassung |
|---|---|
| Referenz-Wrapper | Objekte, die sich wie Referenzen verhalten |
| Smart Pointers | Referenzzählende Smart Pointers |
| Feststellung des Rückgabetyps von Funktionsobjekten | Hilfreich beim Template-Programmieren |
| Verbesserter Member-Zeiger Adapter | 2. Generation mem_fun/mem_fun_ref |
| **Verbesserter Binder** | 2. Generation bind1st/bind2nd |
| **Verallgemeinerte Funktionen** | Verallgemeinerung von Funktionszeigern |
| Type Traits | Reflektion zur Kompilierzeit |
| Zufallszahlen | Unterstützt anpassbare Distributionen |
| Mathematische Sonderfunktionen | Laguerre Polynome, Beta Funktion, usw. |
| Tupel | Verallgemeinerung von pair |
| Arrays mit festen Größen | Arrays mit Elementen der STL Interface |
| Hash-Tabellen | Hash-Tabellen-basierte set/multiset/map/multimap |
| Reguläre Ausdrücke | Verallgemeinerte regex Suchen/Ersetzungen |
| C99 Kompatibilität | 64-Bit ints, <cstdint>, usw. |


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Das Dokument TR1

TR1 ist eine S*pezifikation*:

- Zielt auf Implementierer, nicht Benutzer.
- Enthält keine Begründung für die Funktionalität, die sie spezifiziert.
- Ist nicht isoliert zu betrachten.
  ➧ Setzt C++03-Kenntnisse voraus.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# Das Dokument TR1

Um TR1 wirklich zu *verstehen*:

- Verwenden Sie die weiterführenden Informationen.
- Schauen Sie sich die Erweiterungsvorschläge an.
  - ➧ Links stehen auf der Scott Meyers TR1 Information Website, http://www.aristeia.com/EC3E/TR1_info.html, zur Verfügung.

| EC++ Page | *Effective C++, Third Edition* Name | TR1 Name | Proposal Document |
|---|---|---|---|
| 265 | Smart Pointers | Smart Pointers | n1450 |
| 265 | tr1::function | Polymorphic Function Wrappers | n1402 |
| 266 | tr1::bind | Function Object Binders | n1455 |
| 266 | Hash Tables | Unordered Associative Containers | n1456 |
| 266 | Regular Expressions | Regular Expressions | n1429 |
| 266 | Tuples | Tuple Types | n1403 (PDF) |
| 267 | tr1::array | Fixed Size Array | n1479 |
| 267 | tr1::mem_fn | Function Template mem_fn | n1432 |
| 267 | tr1::reference_wrapper | Reference Wrappers | n1453 |
| 267 | Random Number Generation | Random Number Generation | n1452 |
| 267 | Mathematical Special Functions | Mathematical Special Functions | n1422 |
| 267 | C99 Compatibility Extensions | C Compatibility | n1568 |
| 267 | Type Traits | Metaprogramming and Type Traits | n1424 |
| 267 | tr1::result_of | Function Return Types | n1454 |


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# Was ist Boost?

- Eine Organisation Freiwilliger und eine Website (boost.org).
- Eine Sammlung von C++ Bibliotheken mit diesen Eigenschaften:
  - ➧ Open Source
  - ➧ Portabel
  - ➧ Begutachtet
  - ➧ Von einer "nicht-viralen" Lizenz gedeckt.
- Ein Ort, wo man mögliche Beiträge für die C++ Standardbibliothek erproben kann.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Boost und TR1

Boost hat die meisten Funktionalitäten in TR1 motiviert und alle implementiert:

- Boost Bibliotheken sind lauffähig, TR1 nicht.

TR1 ist völlig oder teilweise auch von anderen Anbietern implementiert:

- Microsoft:
  - ➡ 12/14 Bibliotheken sind im SP1 für englische Ausgaben von VC++ 2008 (VC9).
    - ◆ Nicht verfügbar für "Express" Ausgaben.
  - ➡ Die gleichen Bibliotheken werden mit VC++ 2010 (VC10) geliefert.
- Dinkumware: gesamte TR1 Implementierungen für einige Plattformen.
- Gnu: 10/14 Bibliotheken werden mit gcc 4 geliefert.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## TR1 und Boost

Boost ≠ TR1:

- Boost bietet *viel* mehr Funktionalitäten an als TR1.
  - ➡ Die Bibliotheken sorgen sich selten um embedded Sachen.
    - ◆ Aber gute Leistung ist immer ein Ziel.
- Die Boost APIs sind nicht immer gleich wie in TR1.
  - ➡ z.B. sind die Platzhalter für Bind und Tuple im nicht standardmässigen Namespace.
- Andere TR1 Implementierungen dürfen anders als Boost sein.
  - ➡ TR1 spezifiziert *Interfaces*, nicht Implementierungen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Boost/TR1 Zusammenfassung

- TR1 ist eine Spezifikation für eine Erweiterung der Standardbibliothek.
- Boost ist die führende Quelle für Bibliotheken, die Open Source, portabel und begutachtet sind.
- Viele TR1 Funktionalitäten sind verfügbar von Boost und anderen.
- Boost bietet auch eine Menge zusätzlicher Bibliotheken an.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Überblick

Tag 2 (voraussichtlich):

- Modellierung von memory-mapped I/O
- Implementierung von Callbacks für C Interfaces
- Interessante Anwendungen von Templates:
  - ➧ Typsichere void*-basierte Container
  - ➧ Analyse von physikalischen Maßeinheiten zur Kompilierzeit
  - ➧ Die Spezifikation von endlichen Automaten (falls die Zeit reicht)
- Überlegungen für sicherheitskritische und Echtzeitsysteme
- Weiterführende Informationen (Englisch)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Wie Templates gewöhnliche Casts eliminieren können

Schauen Sie sich dieses Stack Klassen-Template an:

```
template<typename T>
class Stack {
public:
  Stack();
  ~Stack();
  void push(const T& object);
  T pop();

private:
  ...
};
```

Jeder Typ T erzeugt eine neue Klasse.

- Das könnte duplizierten Code verursachen.
- Möglicherweise können Sie solchen Code Bloat nicht akzeptieren.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Eine generische Stack-Klasse für Zeiger

Eine Klasse, die void*-Zeiger nutzt, kann alle (Zeiger-)Stacks darstellen:

```
class GenericPtrStack {
public:
  GenericPtrStack();
  ~GenericPtrStack();

  void push(void *object);
  void * pop();

private:
  ...
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Eine generische Stack-Klasse für Zeiger

GenericPtrStack ist gut, um Code gemeinsam zu benutzen:

```
GenericPtrStack stringPtrStack;
GenericPtrStack intPtrStack;

std::string *newString = new std::string;
int *newInt = new int;

stringPtrStack.push(newString);                    // diese führen den
intPtrStack.push(newInt);                          // gleichen Code aus
```

Aber die Klasse ist leicht falsch anzuwenden:

```
stringPtrStack.push(newInt);                       // oh oh...

std::string *sp =
  static_cast<std::string*>(intPtrStack.pop());    // auch oh oh ...
```

Code gemeinsam zu benutzen ist wichtig, aber ebenso Typsicherheit.

- Wir wollen beides.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Typsichere Interfaces

Man kann Stack partiell spezialisieren, um void*-basierte Klassen zu erzeugen:

```
template<typename T>
class Stack<T*> {
public:
  void push(T *ptr) { s.push(ptr); }

  T * pop() { return static_cast<T*>(s.pop()); }

private:
  GenericPtrStack s;                               // Implementierung
};
```

Stack<T*> Instantiierungen beanspruchen zur Laufzeit *nichts*.

- Alle Instantiierungen nutzen den Code der einzigen GenericPtrStack Klasse
- Alle Stack<T*> Member-Funktionen sind implizit inline

Der Aufwand für Typsicherheit ist *nichts*.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Typsichere Interfaces

Wie zwingt man Programmierer, nur die typsicheren Klassen zu nutzen?

Alles in GenericPtrStack protected machen, damit direktem Einsatz vorgebeugt ist:

```
class GenericPtrStack {
protected:
  GenericPtrStack();
  ~GenericPtrStack();

  void push(void *object);
  void * pop();

private:
  ...                                  // gleich wie zuvor
};

GenericPtrStack stringStack;           // Fehler!
GenericPtrStack intStack;              // Fehler!
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Typsichere Interfaces

Aber jetzt kompiliert Stack<T*> nicht:

```
template<typename T>
class Stack<T*> {
public:
  void push(T *ptr)
  { s.push(ptr); }                   // Fehler! GenericPtrStack::push
                                     // ist protected
  ...

private:
  GenericPtrStack s;
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Typsichere Interfaces

Private Vererbung erlaubt Zugriff auf protected Elemente:

```
template<typename T>
class Stack<T*>: private GenericPtrStack {
public:
  void push(T *objectPtr)
  { GenericPtrStack::push(objectPtr); }

  T * pop()
  { return static_cast<T*>(GenericPtrStack::pop()); }
};
```

Nettoergebnis:

- Maximale Typsicherheit
- Maximale Effizienz

Wie wurde das erreicht?

- void* Zeiger
- Templates
- Private Vererbung
- Inline-Funktionen
- Protected Elemente


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Zusammenfassung: Das Eliminieren von gewöhnlichen Casts

- Templates können typsichere Wrappers um typunsicheren Code erzeugen.
- Mit Inline-Wrapper-Funktionen kann es keinen Laufzeitaufwand geben.
- Sorgfältige Implementierungsentscheidungen können Entwurfsziele sichern.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.

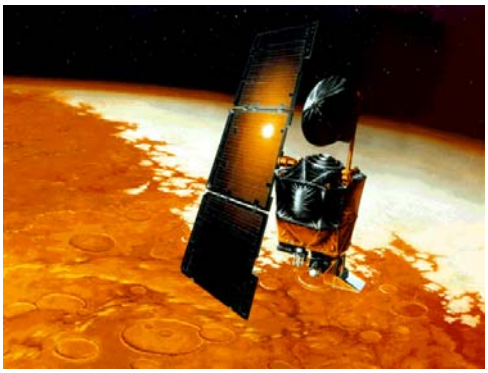


© 2011 Scott Meyers, alle Rechte vorbehalten.

## Sicherstellen dass Dimensionseinheiten richtig sind

Sehr wichtig, die richtigen Einheiten anzuwenden:

- Sinnlos, Zeit und Länge zu vergleichen oder einander zuzuweisen.
- Sinnlos, Pound-Force und Newton zu vergleichen oder einander zuzuweisen.
  - ➧ 1,00 Pound-Force ≅ 4,45 Newton.
  - ➧ Verlust des NASA Mars Climate Orbiters, September 1999.




Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Sicherstellen dass Dimensionseinheiten richtig sind

Leider ignoriert Software oftmals die Einheiten:

```
double t;                       // Zeit (Sekunden)
double a;                       // Beschleunigung (Meter/Sek²)
double d;                       // Länge (Meter)
...
std::cout << d/(t*t) - a;       // okay, subtrahiert Meter/Sek²
std::cout << d/t - a;           // unsinnig, kompiliert aber
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Sicherstellen dass Dimensionseinheiten richtig sind

Das Problem wird durch Typedefs lediglich verdeckt:

```
typedef double Acceleration;
typedef double Time;
typedef double Distance;

Time t;
Acceleration a;
Distance d;

...

std::cout << d/t - a;          // immer noch unsinnig, immer noch akzeptiert
```

Ziel: das Typsystem von C++ benutzen, um Folgendes zu erreichen:

- Einheitenfehler während der Kompilationszeit zu erkennen.
- So wenig Laufzeitleistung wie möglich dafür zu investieren.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Sicherstellen dass Dimensionseinheiten richtig sind

Bemerkungen:

- Dimensionale Typen werden durch dimensionale Exponenten festgelegt:
  - ➧ Velocity = $distance^1/time^1$ = $distance^1 * time^{-1}$
  - ➧ Acceleration = $distance^1/time^2$ = $distance^1 * time^{-2}$
  - ➧ Time = $distance^0 * time^1$
- Jede Kombination von Exponenten ist ein einzigartiger Typ.
  - ➧ Im Prinzip ist die Zahl der Kombinationen unbegrenzt.
- Templates erzeugen Typen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Sicherstellen dass Dimensionseinheiten richtig sind

```
template<int m,                    // Exponent für Masse
         int d,                    // Exponent für Länge
         int t>                    // Exponent für Zeit
class Units {
public:
  explicit Units(double initVal = 0): val(initVal) {}
  double value() const { return val; }
  ...
private:
  double val;
};
```

Jetzt können wir schreiben:

```
Units<1, 0, 0> m;                  // m hat Typ Masse
Units<0, 1, 0> d;                  // d hat Typ Länge
Units<0, 0, 1> t;                  // t hat Typ Zeit

m = t;                             // Fehler! Verschiedene Typen
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Typedefs zum kosmetischen Zweck

Typedefs können die hässlichen Namen verbergen:

```
typedef Units<1, 0, 0> Mass;
typedef Units<0, 1, 0> Distance;
typedef Units<0, 0, 1> Time;

Mass m;
Distance d;
Time t;

m = t;                             // immer noch ein Fehler
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Sicherstellen dass Dimensionseinheiten richtig sind

Arithmetische Operationen sind für solche Typen wichtig, daher:

```
template<int m, int d, int t>
class Units {
public:
  ...                                                    // wie früher
  Units<m, d, t>& operator+=(const Units<m, d, t>& rhs)
  {
    val += rhs.val;
    return *this;
  }
  Units<m, d, t>& operator*=(double rhs)
  {
    val *= rhs;
    return *this;
  }
  ...
};
```

Operatoren für Subtrahieren und Dividieren sind analog.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Sicherstellen dass Dimensionseinheiten richtig sind

Die Nicht-Zuweisungsoperatoren sind vorzugsweise freie Funktionen:

```
template<int m, int d, int t>
Units<m, d, t> operator+(const Units<m, d, t>& lhs,
                         const Units<m, d, t>& rhs)
{
  Units<m, d, t> result(lhs);
  return result += rhs;
}
template<int m, int d, int t>
Units<m, d, t> operator*(double lhs,
                         const Units<m, d, t>& rhs)
{
  Units<m, d, t> result(rhs);
  return result *= lhs;
}
template<int m, int d, int t>
Units<m, d, t> operator*(const Units<m, d, t>& lhs,
                         double rhs)
{
  Units<m, d, t> result(lhs);
  return result *= rhs;
}
```

operator- und operator/ sind ähnlich definiert.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Das Schaffen von typisierten Werten

Nützliche (typisierte) Konstanten aus dem SI-System:

```
const Mass kilogram(1);                 // auf jeden Fall wird der interne
const Distance meter(1);                // double auf 1,0 gesetzt
const Time second(1);
```

Andere nützliche (typiserte) Konstanten sind leicht zu definieren:

```
const Mass pound(kilogram/2.2);         // Avoirdupois-Pound
const Time minute(60 * second);
const Distance inch(.0254 * meter);     // Zoll
```

Wie auch Variablen:

```
int rawLength;                          // untypisierte Länge von äusserer
std::cin >> rawLength;                  // Quelle, als Zoll bekannt
Distance length(rawLength * inch);      // typisierte Länge
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Sicherstellen dass Dimensionseinheiten richtig sind

Der Schlüssel der Methode ist die Multiplikation/Division von Einheiten:

```
template<int m1, int d1, int t1,
         int m2, int d2, int t2>
Units<m1+m2, d1+d2, t1+t2>
operator*(const Units<m1, d1, t1>& lhs,
          const Units<m2, d2, t2>& rhs)
{
  typedef Units<m1+m2, d1+d2, t1+t2> ResultType;
  return ResultType(lhs.value() * rhs.value());
}

template<int m1, int d1, int t1,
         int m2, int d2, int t2>
Units<m1-m2, d1-d2, t1-t2>
operator/(const Units<m1, d1, t1>& lhs,
          const Units<m2, d2, t2>& rhs)
{
  typedef Units<m1-m2, d1-d2, t1-t2> ResultType;
  return ResultType(lhs.value() / rhs.value());
}
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Sicherstellen dass Dimensionseinheiten richtig sind

Echte Implementierungen haben normalerweise mehr Template-Argumente für Einheiten:

- Eins für die Genauigkeit des Werts (in der Regel float oder double)
- Die anderen für die Exponenten der sieben SI-Einheiten:
  - Masse
  - Länge
  - Zeit
  - Stromstärke
  - Temperatur
  - Lichtstärke
  - Stoffmenge


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Sicherstellen dass Dimensionseinheiten richtig sind

```
template<class T, int m, int d, int t, int q, int k, int i, int a>
class Units {
public:
  explicit Units(T initVal = 0) : val(initVal) {}
  T& value() { return val; }
  const T& value() const { return val; }
  ...
private:
  T val;
};

template<class T, int m1, int d1, int t1, int q1, int k1, int i1, int a1,
                  int m2, int d2, int t2, int q2, int k2, int i2, int a2>
Units<T, m1+m2, d1+d2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>
operator*(const Units<T, m1, d1, t1, q1, k1, i1, a1>& lhs,
          const Units<T, m2, d2, t2, q2, k2, i2, a2>& rhs)
{
  typedef Units<T, m1+m2, d1+d2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>
          ResultType;

  return ResultType(lhs.value() * rhs.value());
}
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Dimensionslose Größen

Dimensionslose Größen (d.h., Objekte mit Typ Einheit<T, 0,0,0,0,0,0,0>) sollen kompatibel mit einheitslosen Typen (z.B. int, double, usw.) sein.

- Partielle Template-Spezialisierung hilft:

```
template<typename T>
class Units<T, 0, 0, 0, 0, 0, 0, 0> {
public:
  ...
  Units(T initVal = 0): val(initVal) {}        // Umwandlungen in/aus
  operator T() const { return val; }           // T Werten sind OK

  Units& operator=(T newVal)                   // Zuweisungen von T
  { val = newVal; return *this; }              // Werten sind OK
  ...

private:
  T value;
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Leistungsstarke Dimensionsanalyse

Aktuelle Implementierungen sind noch raffinierter:

- Unterstützen Brüche in Exponenten (z.B. Distance$^{1/2}$)
- Bieten weitere Einheitssysteme an (über SI hinaus)
- Verwenden Template-Metaprogrammierung für Berechnungen zur Kompilationszeit.
  - ➡ z.B. um ggTs zu berechnen, um Exponenten mit Brüchen zu kürzen:
    - ◆ Distance$^{1/2}$ = Distance$^{4/8}$


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Leistungsstarke Dimensionsanalyse

Sie können feststellen, ob diese "einfache Formel,"

$$\frac{1}{X_0} = 4\,\alpha\,r_e^2\,\frac{N_A}{A}\,\left\{Z^2\,\left[L_{rad} - f(Z)\right] + Z\,L'_{rad}\right\}$$

durch diesen C++ Code richtig umgesetzt wird:

```
Energy<> finalEnergy(Element<> const & material, Density<> const dens,
                     Length<> const thick, Energy<> const initEnergy) {
  AtomicWeight<> const A = material->atomicWeight;
  AtomicNumber<> const Z = material->atomicNumber;

  Number<> const L_rad = log( 184.15 / root<3>( Z ) );
  Number<> const Lp_rad = log( 1194. / root<3>(Z*Z) );

  Length<> const X_0 = 4.0 * alpha * r_e * r_e * N_A / A *
                       ( Z * Z * L_rad + Z * Lp_rad );

  return initEnergy / exp( thick / X_0 );
}
```

(Leider nicht. Es gibt drei Einheitenfehler.)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Nicht komplett narrensicher

Einige Kombinationen von Einheiten sind nicht eindeutig.

- Energie und Drehmoment sind beide Länge * Kraft.

Unser Ansatz unterscheidet sie nicht:

```
typedef Units<1, 2, -2> Energy;
typedef Units<1, 2, -2> Torque;

Energy e;
Torque t;

…
e = t;                                  // unsinnig, aber akzeptiert
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

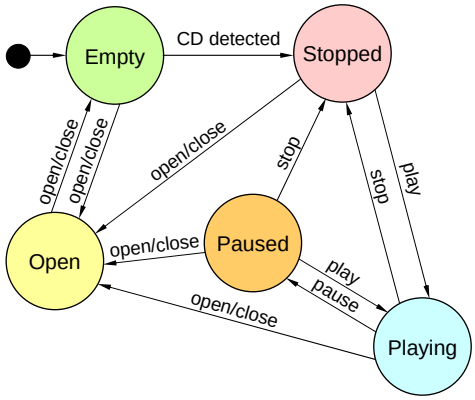
# Zusammenfassung: Sicherstellen dass Dimensionseinheiten richtig sind

- Durch Templates kann man neue Arten von Typsicherheit erreichen.
- Nicht-Typ-Template Parameter sind mächtig und nützlich.
- Templates können zusätzliche Typsicherheit mit geringem oder gar keinem Ressourcenaufwand anbieten.

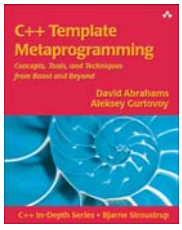

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# Die Spezifikation von Endlichen Automaten (EAs)

Schauen Sie diesen EA für einen CD-Spieler an:




Entnommen (wie das ganze EA Beispiel) aus *C++ Template Metaprogramming* von Abrahams und Gurtovoy.



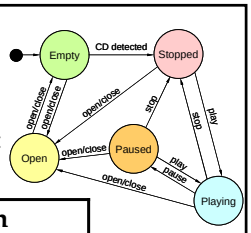

- Komplette Quelle in weiterführenden Informationen.
- Ich habe ihr Material für diese Präsentation leicht abgeändert.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# Die Spezifikation von EAs




Hier eine Tabellen-Version, die auch die Zustandsübergänge zeigt:

| Current State | Event | Next State | Transition Action |
|---|---|---|---|
| Empty | Open/Close | Open | Open drawer |
| Empty | CD-Detected | Stopped | Store CD info |
| Stopped | Play | Playing | Start playback |
| Stopped | Open/Close | Open | Open Drawer |
| Open | Open/Close | Empty | Close drawer; collect CD info |
| Paused | Play | Playing | Resume playback |
| Paused | Stop | Stopped | Stop playback |
| Paused | Open/Close | Open | Stop playback; open drawer |
| Playing | Stop | Stopped | Stop playback |
| Playing | Pause | Paused | Pause playback |
| Playing | Open/Close | Open | Stop playback; open drawer |


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# Domänenspezifische Sprachen

Sowohl Schaubilder als auch Tabellen sind *deklarative Spezifikationen* von EAs.

- Sie spezifizieren, *was* passieren soll, nicht *wie*.

**Template-Metaprogrammierung (TMP) ermöglicht, solche Spezifikationen in C++ auszudrücken.**

- **Durch domänenspezifische eingebettete Sprachen.**
  - ➧ Domänenspezifische Sprachen, die in C++ eingebettet sind.
  - ➧ Englisch: Domain-Specific Embedded Languages (DSELs).
- Mit DSELs sind *Spezifikationen* die Programme.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Die EA DSEL

Für die EA DSEL,

- Clients detaillieren:
  - States (Zustände)
  - Events (Ereignisse)
  - Transitions (Zustandsübergänge)
  - Transition actions (Übergangsaktionen)

Der Compiler erstellt den Code für den EA dann automatisch.

- Durch Template-Instantiierung.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die EA DSEL

In diesem Vortrag betrachten wir nur das **Client-Interface** des EAs:

- Wie Clients eine TMP-basierte EA-Bibliothek *benutzen*.
  - Ziel: demonstrieren, was man erreichen kann.
- Die Umsetzung der Bibliothek steht in *C++ Template Metaprogramming*.
  - Sie ist das Kapitel 11 in einem 11-Kapitel Buch.
    - Uns fehlt die Zeit, Kapitel 1-10 abzudecken :-)

Nur grundsätzliche Funktionalität wird gezeigt.

- Keine Zustand-Eingangs/Ausgangsaktionen, keine Wächter, keine Zustandshierarchien, usw.
- **Ziel ist, nicht-offensichtliche Template-Funktionalität zu zeigen,** nicht zu erklären, wie EAs umgesetzt werden können.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Das Spezifizieren von EAs und Zuständen

Ein EA wird als Klasse dargestellt, seine Zustände als geschachtelter enum:

```
class player : public state_machine<player>          // player = EA für
{                                                    // den CD-Spieler
  private:
    enum states {                                    // Zustände im EA
      Empty, Open, Stopped, Playing, Paused
      , initial_state = Empty                        // initialer Zustand
    };                                               // des EAs
  …
};
```

- Basisklasse state_machine bietet generische EA-Funktionalität.
  - ➡ TMP-erzeugter Code wird in diese Klasse gestellt.
  - ➡ Basisklasse nimmt abgeleitete Klasse als Template-Parameter.
    - ◆ Muster: “The Curiously Recurring Template Pattern”
- Zustände sind private.
  - ➡ EA-Clients brauchen nicht, auf sie zuzugreifen.
    - ◆ EA-Clients erstellen bloß Ereignisse.
    - ◆ Der EA reagiert passend.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## EA-Ereignisse spezifizieren

Ereignisse sind Klassen. In diesem Beispiel

- werden sie ausserhalb der EA-Klasse definiert.
  - ➡ Es wäre auch möglich, sie in die Klasse zu schachteln.
- sind sie meistens leer.
  - ➡ In einem echten System können sie beliebig kompliziert sein.

```
struct play {};
struct open_close {};
struct pause {};
struct stop {};
class cd_detected {
public:
  cd_detected(char const* cdName,
              std::vector<std::clock_t> const& trackLengths) { … }
  …
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# EA-Übergangsaktionen spezifizieren

Übergangsaktionen sind EA Member-Funktionen:

```
class player : public state_machine<player>
{
private:
  enum states { … };

  void start_playback(play const&);
  void open_drawer(open_close const&);
  void close_drawer(open_close const&);
  void store_cd_info(cd_detected const&);
  void stop_playback(stop const&);
  void pause_playback(pause const&);
  void resume_playback(play const&);
  void stop_and_open(open_close const&);

  …
};
```

- Wie Zustände sind Übergangsaktionen private.
  - ➧ EA-Clients erstellen bloß Ereignisse.
  - ➧ Der EA reagiert automatisch.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


# EA-Übergänge spezifizieren

Die Struktur des EAs ist als eine Tabelle spezifiziert.

- Basisklasse state_machine deklariert ein row-Template:

```
template<class Derived>
class state_machine
{
...
protected:
  template<
      int CurrentState
    , class Event
    , int NextState
    , void (Derived::*action)(Event const&)
  >
  struct row { … };

…
};
```

| Current State | Event | Next State | Transition Action |
|---|---|---|---|
| Empty | Open/Close | Open | Open drawer |
| Empty | CD Detected | Stopped | Store CD info. |
| Stopped | Play | Playing | Start playback |
| Stopped | Open/Close | Open | Open Drawer |
| Open | Open/Close | Empty | Close drawer; collect CD info. |
| Paused | Play | Playing | Resume playback |
| Paused | Stop | Stopped | Stop playback |
| Paused | Open/Close | Open | Stop playback; open drawer |
| Playing | Stop | Stopped | Stop playback |
| Playing | Pause | Paused | Pause playback |
| Playing | Open/Close | Open | Stop playback; open drawer |


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## EA-Übergänge spezifizieren

Der EA beschreibt die Tabelle als eine Sammlung von rows:

```
class player : public state_machine<player> {
private:
  …                                  // Zustände und Übergänge
  typedef player p;                  // damit die Tabelle leichter lesbar ist
  struct transition_table : mpl::vector11<
   //    Start     Event         Next      Action
   //  +---------+-------------+---------+---------------------+
   row < Stopped , play        , Playing , &p::start_playback  >,
   row < Stopped , open_close  , Open    , &p::open_drawer     >,
   //  +---------+-------------+---------+---------------------+
   row < Open    , open_close  , Empty   , &p::close_drawer    >,
   //  +---------+-------------+---------+---------------------+
   …
  > {};
};
```

- Ein Font mit fester Zeichenbreite lässt die Tabelle leichter erkennen.
- mpl::vector11 ist ein vector-ähnlicher TMP-Container mit 11 Typen.
  - ➧ In diesem Fall, 11 row-Instantiierungen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## EA-Übergänge spezifizieren

Die komplette Tabelle:

```
class player : public state_machine<player> {
…
struct transition_table : mpl::vector11<
  //    Start     Event         Next      Action
  //  +---------+-------------+---------+---------------------+
  row < Empty   , open_close  , Open    , &p::open_drawer     >,
  row < Empty   , cd_detected , Stopped , &p::store_cd_info   >,
  //  +---------+-------------+---------+---------------------+
  row < Stopped , play        , Playing , &p::start_playback  >,
  row < Stopped , open_close  , Open    , &p::open_drawer     >,
  //  +---------+-------------+---------+---------------------+
  row < Open    , open_close  , Empty   , &p::close_drawer    >,
  //  +---------+-------------+---------+---------------------+
  row < Paused  , play        , Playing , &p::resume_playback >,
  row < Paused  , stop        , Stopped , &p::stop_playback   >,
  row < Paused  , open_close  , Open    , &p::stop_and_open   >,
  //  +---------+-------------+---------+---------------------+
  row < Playing , stop        , Stopped , &p::stop_playback   >,
  row < Playing , pause       , Paused  , &p::pause_playback  >,
  row < Playing , open_close  , Open    , &p::stop_and_open   >
  //  +---------+-------------+---------+---------------------+
 > {};
};
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Ein EA in C++ vs. ein EA in einer Tabelle

Vergleichen Sie das mit der ursprünglichen Tabelle:

| Current State | Event | Next State | Transition Action |
|---|---|---|---|
| Empty | Open/Close | Open | Open drawer |
| Empty | CD-Detected | Stopped | Store CD info |
| Stopped | Play | Playing | Start playback |
| Stopped | Open/Close | Open | Open Drawer |
| Open | Open/Close | Empty | Close drawer; collect CD info |
| Paused | Play | Playing | Resume playback |
| Paused | Stop | Stopped | Stop playback |
| Paused | Open/Close | Open | Stop playback; open drawer |
| Playing | Stop | Stopped | Stop playback |
| Playing | Pause | Paused | Pause playback |
| Playing | Open/Close | Open | Stop playback; open drawer |

- Die Tabelle *ist* der Quellcode!


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Die Verwendung von EAs

Client-Code muss nur Ereignisse erzeugen:

- Noch einmal zitiere ich *C++ Template Metaprogramming*.

```
player p;                                   // An instance of the FSM

p.process_event(open_close());              // user opens CD player
p.process_event(open_close());              // inserts CD and closes
p.process_event(                            // CD is detected
   cd_detected(  "louie, louie"
               , std::vector<std::clock_t>( /* track lengths */ ))
);

p.process_event(play());                    // etc.
p.process_event(pause());
p.process_event(play());
p.process_event(stop());
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Zusammenfassung: Die Spezifikation von EAs

- Template-Metaprogrammierung ermöglicht die Schaffung von Domain-Specific Embedded Languages (DSELs).
- DSELs erleichtern einen Stil des deklarativen Programmierens.
- In der Regel ist deklarativer Code leichter zu schreiben, zu verstehen und weiterzuentwickeln.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Andere EA-Ansätze

Es gibt viele Wege, EAs zu spezifizieren und umzusetzen. Sie unterscheiden sich in:

- **Ausdrucksfähigkeit**, z.B. Unterstützung für hierarchische und nebenläufige Zustände.
  ➧ UML bietet beides (etwa Statecharts + Petri-Netz)
- Möglichkeit, **EAs statisch als auch dynamisch zu definieren.**
- **Typsicherheit** des Quellcodes.
- Anforderung an **Template-Unterstützung des Compilers**.
- **Größe und Geschwindigkeit** des ausführbaren Codes.
- Die Menge **Heap-Speicher**, den sie beanspruchen.
- Unterstützung für **Multithreading**.
- **Debugfähigkeit**: wie leicht, sich der EA debuggen lässt.
- **Kopplung** zwischen z.B. Zuständen und Zustandsübergängen.

Viele Ansätze sind in den weiterführenden Informationen beschrieben.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Zusammenfassung: Interessante Anwendungen von Templates

- **Templates sind für weit mehr nützlich als nur Container.**
- Templates können typsichere Wrappers um typ-unsicheren Code erzeugen.
- Templates können neue Arten von Typsicherheit erstellen (z.B. dimensionale Einheiten).
- Domain-Specific Embedded Languages (DSELs) können auf Templates aufgebaut werden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Überblick

Tag 2 (voraussichtlich):

- Modellierung von memory-mapped I/O
- Implementierung von Callbacks für C Interfaces
- Interessante Anwendungen von Templates:
  - ➧ Typsichere `void*`-basierte Container
  - ➧ Analyse von physikalischen Maßeinheiten zur Kompilierzeit
  - ➧ Die Spezifikation von endlichen Automaten (falls die Zeit reicht)
- Überlegungen für sicherheitskritische und Echtzeitsysteme
- Weiterführende Informationen (Englisch)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C++ in sicherheitskritschen Systemen

*Sicherheitskritisch*: Systemfehlfunktion ⇒ Tod oder ernsthafte Verletzung.

Einige Anwendungsbereiche:

- **Transport**: Flugzeuge, Autos, Züge, Schiffe, usw.
- **Medizin**: Bestrahlungsgeräte, Herz-Lungen-Maschinen, Infusionsgeräte, usw.
- **Kommunikation**: Militärfunk, Notrufnummern (z.B. 112 in Europa), usw.

Aktuelle Verwendung von C++ in solchen Systemen?

- **Weit verbreitet:** Alle genannten Anwendungsbereiche.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Sicherheitskritsche Software und Risiken

Sicherheitskritsche Software ist wie "normale" Software, ausser:

- Das Risiko für falsches Verhalten muss ausserordentlich niedrig sein.

Der Schlüssel ist daher:

- **Das Riskio für falsches Verhalten minimieren.**


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Risiko minimieren

Ansätze:

- Sehr detaillierte Spezifikationen:
  - ➡ Und strenge Verwaltung von Änderungen.
- Umfassende Tests.
  - ➡ **Auf mehrfachen Ebenen**, z.B. Einheit, Modul, System.
  - ➡ **Einschließlich Performance.**
    - ◆ Ohne ausreichende Performance kann kein System korrekt sein.
- Beschränkte Freiheit der Programmierer.
  - ➡ Durch Programmierrichtlinien.
- Ausführliche statische Analyse:
  - ➡ Sicherstellen, dass Programmierrichtlinien befolgt wurden.
  - ➡ Suchen nach Problemen, die Tests nur schwer finden können.
  - ➡ Analysen durch Mensch und Maschine.
    - ◆ Tools wie Lint.
    - ◆ Formale Code-Inspektionen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Risiko minimieren

- Unabhängige redundante Berechnungen:
  - ➡ Unabhängige Teams implementieren dieselbe Funktionalität.
    - ◆ Möglicherweise in verschiedenen Programmiersprachen.
  - ➡ Alle Implementierungen laufen gleichzeitig.
  - ➡ Falls es verschiedene Ergebnisse gibt,
    - ◆ Abstimmen?
    - ◆ Runterfahren?
    - ◆ Auf einen bekannten Zustand zurückfallen?


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Risiko minimieren

Also:

- Detaillierte Spezifikationen.
- Umfassende Tests.
- Beschränkte Programmiererfreiheit.
- Ausführliche statische Analyse.
- Unabhängige redundante Berechnungen.

Nichts davon ist C++-spezifisch:

- *Entwicklungsprozess* ist wesentlich wichtiger als Programmiersprache.
- Das einzige, was einzigartig für C++ ist, sind die Programmierrichtlinien.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Programmierrichtlinien

Ziele:

- Code-Klarheit und Verständlichkeit maximieren.
  - ➧ Für Menschen als auch für Tools.
- Vorhersagbarkeit des Code-Verhaltens maximieren.

Mittel:

- Anforderungen und Verbote im Coding-Bereich.

Idealerweise können die Richtlinien automatisch überprüft werden.

- Nicht immer realisierbar.
  - ➧ z.B. merkt Hatton an, dass 5-10% der MISRA-C Richtlinien nicht automatisch überprüfbar sind.
- Menschen müssen Richtlinien überwachen, die nicht automatisch prüfbar sind.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Rangordnung von Richtlinien

Richtlinien haben normalerweise mehrere Ränge:

- z.B. haben die Joint Strike Fighter (JSF) Richtlinien drei Ränge:
  - **"Should" (Sollte)**: beratend.
  - **"Will" (Soll)**: pflichtmäßig, Bestätigung nicht gefordert.
  - **"Shall" (Muss)**: pflichtmäßig, Bestätigung gefordert.
- Andere Sätze von Richtlinien unterscheiden pflichtmäßige Regeln von beratenden Regeln, usw.

Niedrigere Ränge entsprechen größerer Programmier-Freiheit.

- Höhere Ränge sind deswegen vorzuziehen.
- Sogar das Verstoßen gegen eine JSF "Sollten"-Regel erfordert Zustimmung eines Managers.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Arten von Richtlinien

**Richtlinien zur Anwendung der Sprache:**

- spezifieren zugelassene Spracheigenschaften.
  - Entfernen "unnötige" und "gefährliche" Teile der Sprache.
    - Identifizieren die akzeptablen Elemente der Sprache.
- reduzieren die Komplexität des Codes.
- machen das Verhalten der Software vorhersagbarer.
- erleichtern statische Analyse.

Richtlinien für C++ basieren oft auf MISRA-C++ oder MISRA-C.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beispiele für Richtlinien

- **"Unterspezifiziertes" Verhalten darf nicht verwendet werden.**
  - Mehrere "offizielle" Begriffe: "undefiniert", "unspezifiziert", usw.
  - Üblich in C++.
    - Aus: *An Investigation of the Unpredicatable Features of the C++ Language*:

| Category | Language issues | Library Issues | Total Number of Issues |
|---|---|---|---|
| Unspecified behaviour | 25 | 25 | 50 |
| Undefined behaviour | 77 | 29 | 106 |
| Implementation behaviour | 58 | 23 | 81 |
| Indeterminate behaviour | 5 | 0 | 5 |
| Behaviour that requires no diagnostic | 18 | 0 | 18 |

    - Beispiel:

```
f(calcThis(), calcThat());      // verboten: Evaluierungsreihenfolge
                                // undefiniert

int thisVal = calcThis();
int thatVal = calcThat();
f(thisVal, thatVal);            // okay, Evaluierungsreihenfolge
                                // wohldefiniert
```


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Beispiele für Richtlinien

- **Vermeiden "überraschendes" Verhalten:**
  - *JSF AV Regel 177:* Benutzerdefinierte Umwandlungsfunktionen sollten vermieden werden.
    - Beugt unerwarteten Umwandlungen vor.
  - *MISRA C++ Regel 5-0-1:* Der Wert eines Ausdrucks muss unter allen erlaubten Evaluierungsreihenfolgen gleich sein.
    - Beugt Compiler- oder optimierungsabhängigem Verhalten vor.
  - *High Integrity C++ Regel 3.3.14:* Deklarieren Sie den Copy-Zuweisungsoperator protected in einer abstrakten Klasse.
    - Beugt teilweisen Zuweisungen auf abgeleitete Objekte durch Zeiger und Referenzen auf Basisklassen vor.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Beispiele für Richtlinien

- **Vermeiden Sie unnötig komplexen Code.**
  - *JSF AV Regel 3:* Alle Funktionen müssen eine zyklomatische Komplexitätszahl von 20 oder weniger haben.
  - *High Integrity C++ Regel 4.1:* Schreiben Sie keine Funktionen, die eine exzessive McCabe zyklomatische Komplexität haben.
    - Empfohlenes Maximum ist 10.
  - *MISRA C++ Regel 6-6-3:* Die `continue` Anweisung darf nur in einer "wohlgeformten" `for`-Schleife benutzt werden.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Risikoverringerung durch Wahl der Sprache

C++ kann das Risiko im Vergleich zu C reduzieren:

- Programmierer arbeiten auf einem höheren Abstraktionsniveau:
  - Der Code sieht dem Entwurf ähnlicher.
  - Unmittelbare Unterstützung von mehreren Paradigmen.
    - **OO**: Klassen, Vererbung, dynamische Bindung, usw.
    - **Generisch**: Templates
    - **Imperativ**
    - **Funktional** (leider gibt es keine Closures oder Lambdas vor C++11)
- Sprachelemente reduzieren den Bedarf, den Präprozessor zu nutzen.
  - z.B. C Makros werden oft zu C++ `const`s und `inline`s.
- Templates bieten typsichere Alternativen zu typunsicheren C Ansätzen.
  - z.B. vermeiden Verwirrung von Objekt- und Array-Zeigern:
    - `std::auto_ptr<T>` oder `std::tr1::shared_ptr<T>` ⇒ einzelnes Objekt.
    - `std::vector<T>` oder `std::tr1::array<T, n>` ⇒ Array von Objekten.

Risiko von C++ im Vergleich zu Java, Ada, C#, usw. heiß umstritten :-)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Tool-bezogene Risiken

Compiler, Linker, Laufzeitsysteme, Betriebssysteme, usw. sind Software.

- Sie tragen auch zur Zuverlässigkeit der Systeme bei.
- Risikoreduzierung bedeutet, ihre Risiken auch anzugehen.

Ansätze:

- **Kommerzielle Validierungssuites:**
  - ➧ z.B. für Übereinstimmung von Compilern/Bibliotheken mit dem Standard für C++.
  - ➧ z.B. für DO-178B.
- **Manuelle Analyse des erzeugten Codes.**
  - ➧ Typischerweise auch mit einem begrenzten Satz von Sprachfeatures.
- **Testen, Testen, Testen.**

C++ Compiler normalerweise nicht zertifiziert.

- Green Hills Compiler für Embedded C++ wurde nach DO-178B Level A zertifiziert.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Zusammenfassung: C++ in sicherheitskritschen Systemen

- Grundsätzlich geht es darum, Risiken zu minimieren.
- Entwicklungsprozess wichtiger als Programmiersprache.
- Programmierrichtlinien und umfassende statische Analyse sind grundlegend.
- Zuverlässigkeit von zugehörigen Tools und Bibliotheken auch wichtig.
- C++ wird oft in sicherheitskritischen Bereichen erfolgreich verwendet.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## C++ in Echtzeitsystemen

*Echtzeit:*

- **Hart**: Zeitschranke überschritten ⇒ System scheitert.
  - ➧ z.B. Motorsteuergeräte, Herzschrittmacher, Aufzüge, usw.
- **Weich**: Zeitschranke überschritten ⇒ niedrigere Qualität.
  - ➧ z.B. Musik- und Videospieler, IP Netzwerk Pufferverwaltung, usw.

Entscheidendes Merkmal ist nicht Geschwindigkeit, sondern Determinismus bezüglich des Zeitverhaltens:

- Echtzeitssysteme gewährleisten, dass sie ihre Zeitbeschränkungen immer oder meistens einhalten.
  - ➧ *Immer* für harte Echtzeit.
  - ➧ *Meistens* für weiche Echtzeit.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Entwicklungsansatz für Echtzeitsysteme

Echtzeit-Entwicklung für C++ im Wesentlichen gleich wie für C:

- Zeitbeschränkungen feststellen.
- C++ Features mit unbestimmtem Zeitverhalten vermeiden:
  - ➧ C: "normale" malloc/free/memcpy
  - ➧ C++:
    - ◆ **"normale" malloc/free/memcpy/new/delete**
    - ◆ **RTTI**: dynamic_cast, Vergleiche von type_info Objekten
    - ◆ **Exceptions**: try/throw/catch
  - ➧ Massgeschneiderte malloc/free/memcpy usw. können bestimmtes Zeitverhalten haben.
- Laufzeitanalyse durchführen.
  - ➧ Für Funktionen, Tasks, und das ganze System.
  - ➧ Harte Echtzeit: normalerweise WCET-Analyse.
    - ◆ WCET = Worst Case Execution Time.
  - ➧ Weiche Echtzeit: oft Analyse der durchschnittlichen Laufzeit.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Variationen im Zeitverhalten

Laufzeit für Sprachfeatures hängt ab von:

- Compiler und Linker.
  - ➧ Einschliesslich der Optimierungseinstellungen
- Aufruf-Kontext (für Inline-Funktionen).
- Eigenschaften der Hardware:
  - ➧ z.B. Caching, Pipelining, spekulative Befehlsausführung, usw.

Zusätzlich:

- Bibliotheksfeatures dürfen auf unterschiedliche Weise umgesetzt werden:
  - ➧ Implementierungen von Containern/Algorithmen in der STL.
    - z.B. std::string darf SSO oder COW verwenden (oder nicht).
    - z.B. std::sort darf quicksort oder introsort verwenden.
  - ➧ Unterschiedliche Algorithmen zur Heapverwaltung für malloc/free/new/delete.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Laufzeitanalyse

Ansätze, Block/Funktion WCET-Analyse durchzuführen:

- **Statische Analyse vom Quellcode.**
  - ➧ Von Menschen, Werkzeugen, oder beiden.
  - ➧ Templates können durch explizite Instantiierung und jeweilige Instantiierungsanalyse behandelt werden.
- **Dynamische Analyse des Codes.**
  - ➧ Beobachten, wie lange Blöcke/Funktionen laufen.
- **Eine Kombination von beidem.**
  - ➧ Dynamische Analyse der WCETs der Blöcke.
  - ➧ Statische Analyse der Pfade durch die Blöcke.
    - Schwierig, 100% Pfadabdeckung zu erreichen.

Für System-WCET, eine Kombination von:

- Pro-Task WCET-Analyse.
- Analyse von den Abfolgemöglichkeiten der Tasks.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Laufzeitanalyse

Ansätze, die durchschnittliche Laufzeit zu analysieren:

- **Gleiche Optionen für WCET.**
  - ➡ Aber durchschnittliche Zeit feststellen statt worst-case.
- **C++ Anweisungszahl mit einem Sicherheitsfaktor multiplizieren.**
  - ➡ Der Sicherheitsfaktor ist größer als für C.
    - ◆ C++ Anweisungen bewirken normalerweise mehr als in C.
  - ➡ Hilfreich, Laufzeit während der Entwicklung abzuschätzen.
    - ◆ Reduziert den Bedarf für spätere kleinere Anpassungen.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Zusammenfassung: C++ in Echtzeitsystemen

- Grundsätzlicher Ansatz gleich wie C.
- Normalerweise wird auf Heap-Operationen, RTTI, und Exceptions verzichtet.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Überblick

Tag 2 (voraussichtlich):

- Modellierung von memory-mapped I/O
- Implementierung von Callbacks für C Interfaces
- Interessante Anwendungen von Templates:
  - ➧ Typsichere void*-basierte Container
  - ➧ Analyse von physikalischen Maßeinheiten zur Kompilierzeit
  - ➧ Die Spezifikation von endlichen Automaten (falls die Zeit reicht)
- Überlegungen für sicherheitskritische und Echtzeitsysteme
- Weiterführende Informationen (Englisch)


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

Using C++ in embedded systems:

- "Abstraction and the C++ Machine Model," Bjarne Stroustrup, Keynote address at ICESS04, December 2004, available at http://www.research.att.com/~bs/abstraction-and-machine.pdf.
  - ➧ An overview of the strengths of C++ for embedded systems.
- "OO Techniques Applied to a Real-time, Embedded, Spaceborne Application," Alexander Murray and Mohammad Shababuddin, *Proceedings of OOPSLA 2006*.
  - ➧ Describes how OO and C++ are being used in the development of satellite software.
- "Reducing Run-Time Overhead in C++ Programs," Embedded Systems Conference, Dan Saks, 1998 and subsequent years.
  - ➧ How to avoid common C++ performance "gotchas".
  - ➧ 2002 paper available at http://www.open-std.org/jtc1/sc22/wg21/docs/ESC_SF_02_405_&_445_paper.pdf


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

More on using C++ in embedded systems:

- "C++ in Embedded Systems: Myth and Reality," Dominic Herity, *Embedded Systems Programming*, February 1998.
  - ➧ Dated, but a good, comprehensive overview of C++ vs. C. Considers code size, code speed, exceptions, ROMability, etc.
- "Embedded Programming with C++," Stephen Williams, Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), 1997.
  - ➧ Summarizes the design, functionality, and performance of a C++ runtime library for embedded systems; the runtime replaces the OS.
- "C++ in der Automotive-Software-Entwicklung," Matthias Kessler, Oliver Müller, and Gerd Schäfer, *Elektronik automotive*, May 2006.
  - ➧ An overview of C++ language features and how they've proven useful in the development of embedded automotive software.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

General information on how C++ is implemented:

- *Technical Report on C++ Performance*, ISO/IEC TR 18015:2006(E), February 2006, http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf.
  - ➧ Summarizes likely overhead for various language features.
- *Inside the C++ Object Model*, Stanley B. Lippman, Addison-Wesley, 1996, ISBN 0-201-83454-5.
  - ➧ Information is sometimes outdated, incorrect, or cfront-specific.
- *Secure Coding in C and C++*, Robert Seacord, Addison-Wesley, 2006, ISBN 0-321-33572-4.
  - ➧ Good treatments of runtime data structures and how undefined behavior can lead to security vulnerabilities.
- "C++ Exceptions and the Linux Kernel," Halldór Ísak Glyfason and Gísli Hjálmtysson, *Dr. Dobbs Journal*, September 2005.
  - ➧ Focuses on exceptions, but mentions other language features, too.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

More general information on how C++ is implemented:

- "Vtbl layout under MI ," comp.lang.c++.moderated thread, initiated 27 May 2005 by Scott Meyers. Available at http://tinyurl.com/rs3rb.
  - ➧ Note gcc's "-fdump-class-hierarchy" option.
- "Exception Handling," Josée Lajoie, *C++ Report*, March-April 1994 (Part 1) and June 1994 (Part 2).
  - ➧ Overview of a table-based EH implementation.
  - ➧ Part 1 was reprinted in *C++ Gems*, Stanley Lippman (Ed.), SIGS Books & Multimedia, 1996, ISBN 1-884842-37-2.
- "Itanium C++ ABI," *CodeSourcery* web site, available at http://www.codesourcery.com/cxx-abi/abi.html.
  - ➧ Includes an ABI specification for table-based EH implementations (for 64 bit applications).
- "Fast Dynamic Casting," Michael Gibbs and Bjarne Stroustrup, *Software Practice & Experience*, December 2005.
  - ➧ Describes a constant-time implementation of dynamic_cast.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

Information on how Microsoft's C++ is implemented:

- "How a C++ Compiler Implements Exception Handling," Vishal Kochhar, *The Code Project* web site, April 2002, available at http://www.codeproject.com/cpp/exceptionhandler.asp.
  - ➧ An excruciatingly detailed description of how EH is implemented in MSVC6-7.
- "The Cost of C++ Exception Handling on Windows," Kevin Frei, Presentation to the *Northwest C++ Users Group*, October 18, 2006.
- "C++ Under the Hood," Jan Gray, in *Black Belt C++: The Masters Collection*, M&T Books, 1994, ISBN 1-55851-334-5
  - ➧ Describes how language features are (were?) implemented in MSVC.
- "Microsoft C++ Name Mangling Scheme," Kang Seonghoon, http://mearie.org/documents/mscmangle/.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

Program behavior when pure virtual functions are called:

- "'Pure Virtual Function Called': An Explanation," *The C++ Source*, February 26, 2007, http://www.artima.com/cppsource/pure_virtual.html.

Cost of error handling without exceptions:

- "Bail, return, jump, or…throw," Dan Saks, *Embedded Systems Design*, March 2007.
  - ➧ Estimates object code size increase of 15-40% for using return values to report error conditions (vs. no error detection/propagation).


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

Controlling the generation and cost of temporary objects:

- *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Scott Meyers, Addison-Wesley, 1996, ISBN 0-201-63371-X.
  - ➧ Items 19-22 cover the basics of controlling temporary creation.
  - ➧ Item 29 describes how reference counting can make object creation inexpensive.
  - ➧ A copy of the book's table of contents is attached.
- Template metaprogramming (TMP), including expression templates and TMP-based libraries:
  - ➧ "Using C++ Template Metaprograms," Todd Veldhuizen, *C++ Report*, May 1995.
  - ➧ *C++ Templates*, David Vandevoorde and Nicolai Josuttis, Addison-Wesley, 2003, ISBN 0-201-73484-2, chapters 17-18.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

Code bloat:

- *Techniques for Scientific C++*, Todd Veldhuizen, Indiana University Computer Science Technical Report # 542, August 2000. Available at http://www.osl.iu.edu/~tveldhui/papers/techniques/.
  - ➧ Section 1.5 is entitled "Managing Code Bloat."
- "Code Bloat due to Templates," comp.lang.c++.moderated thread, initiated 16 May 2002. Available at http://tinyurl.com/xnhn.
- "C++ Templates vs. .NET Generics," comp.lang.c++.moderated thread, initiated 4 October 2003. Available at http://tinyurl.com/xnhf.
  - ➧ The 13 November posting by Mogens Hansen initiates a good subthread on commonality/variability analysis.
- "Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat," Lubomir Bourdev and Jaakko Järvi, *Library-Centric Software Design* (LCSD '06), 22 October 2006.
  - ➧ Describes an advanced code hoisting technique.
- "Minimizing Dependencies withing Generic Classes for Faster and Smaller Programs," Dan Tsafrir *et al.*, OOPSLA '09, October 2009.
  - ➧ Discusses performance costs of unnecessarily template class nesting.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

Inlining:

- "Inline Redux," Herb Sutter, *C/C++ Users Journal*, November 2003.
- "Inline Functions," Randy Meyers, *C/C++ Users Journal*, July 2002.
- *Efficient C++*, Dov Bulka and David Mayhew, Addison-Wesley, 2000, ISBN 0-201-37950-3.
  - ➧ *Three chapters* on getting the most out of inlining!
- "Link-Time Code Generation," Matt Pietrek, *MSDN Magazine*, May 2002.
  - ➧ Describes link-time inlining in Visual C++ .NET, including how the system determines which functions will be inlined.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

Using link-time polymorphism:

- "Effective Test-Driven Development for Embedded Software," Michael Karlesky *et al.*, IEEE 2006 Electro/Information Technology Conference, May 2006.
  - ➧ Uses link-time polymorphism to achieve TDD for C.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

ROMing objects:

- "Static vs. Dynamic Initialization," Dan Saks, *Embedded Systems Programming*, December 1998 and "Ensuring Static Initialization in C++," *Embedded Systems Programming*, March 1999.
  - ➧ Summarizes when compilers are most likely to ROM data.
- *Technical Report on C++ Performance*, ISO/IEC TR 18015:2006(E).
  - ➧ Discusses what can be ROMed, at least in theory.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

Memory management in embedded systems:

- *Small Memory Software*, James Noble and Charles Weir, Addison-Wesley, 2001, ISBN 0-201-59607-5, chapter 5.
  ➧ Also explores other topics related to embedded systems software.
- *Real-Time Design Patterns*, Bruce Powel Douglass, Addison-Wesley, 2003, ISBN 0-201-69956-7, chapter 6.
- "Improving Performance for Dynamic Memory Allocation," Marco Varlese, *Embedded Systems Design*, May 2009.
  ➧ Describes implementation of a block allocator.

Guidelines for understanding, using, and writing new and delete:

- *Effective C++, Third Edition*, Scott Meyers, Addison-Wesley, 2005, ISBN 0-321-33487-6.
- *Effective C++, Second Edition*, Scott Meyers, Addison-Wesley, 1998, ISBN 0-201-92488-9.
- *More Effective C++*, Scott Meyers, Addison-Wesley, 1996, ISBN 0-201-63371-X.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

General information on new, delete, and memory management:

- "Efficient Memory Allocation," Sasha Gontmakher and Ilan Horn, *Dr. Dobbs Journal*, January 1999, pp. 116ff.
- *Modern C++ Design*, Andrei Alexandrescu, Addison-Wesley, 2001, ISBN 0-201-70431-5, chapter 4.
- "Memory Management & Embedded Databases," Andrei Gorine and Konstantin Knizhnik, *Dr. Dobbs Journal*, December 2005.
- "Boost Pool Library," Stephen Cleary, http://www.boost.org/libs/pool/doc/index.html.
- "A Memory Allocator," Doug Lea, http://gee.cs.oswego.edu/dl/html/malloc.html.
- "Scalable Lock-Free Dynamic Memory Allocation," Maged M. Michael, *Proceedings of the 2004 Conference on Programming Language Design and Implementation (PLDI)*, available at http://www.cs.utah.edu/~wilson/compilers/papers/pldi04-michael.pdf.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

STL allocators:

- *Effective STL*, Scott Meyers, Addison-Wesley, 2001, ISBN 0-201-74962-9.
- *The C++ Programming Language (Third Edition)*, Bjarne Stroustrup, Addison-Wesley, 1997, ISBN 0-201-88954-4, pp. 567-576.
- "Custom STL Allocators," Pete Isensee, http://www.tantalon.com/pete/customallocators.ppt.
  - ➧ PPT materials from a 2003(?) Game Developers Conference Talk.
- "Improving Performance with Custom Pool Allocators for STL," Anthony Aue, *Dr. Dobbs Journal*, September 2005.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

Modeling memory-mapped IO:

- "Register Access in C++," Pete Goodliffe, *C/C++ Users Journal*, May 2005.
- Columns by Dan Saks in *Embedded Systems Programming*, *Embedded Systems Design*, or at embedded.com:
  - ➧ "Mapping Memory," September 2004.
  - ➧ "Mapping Memory Efficiently," November 2004.
  - ➧ "More Ways to Map Memory," January 2005.
  - ➧ "Sizing and Aligning Device Registers," May 2005.
  - ➧ "Alternative models for memory-mapped devices," May 2010.
  - ➧ "Memory-mapped devices as C++ classes," June 22, 2010.
  - ➧ "Accessing memory-mapped classes directly," September 2010.
  - ➧ "Bundled vs. unbundled monostate classes," November 11, 2010.
  - ➧ "Measuing instead of speculating," December 2010.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

More on Modeling memory-mapped IO:

- "Representing and Manipulating Hardware in Standard C and C++," Embedded Systems Conference, Dan Saks, 1999 and subsequent years.
  - 2002 paper available at http://www.open-std.org/jtc1/sc22/wg21/docs/ESC_SF_02_465_paper.pdf.
- *Technical Report on C++ Performance*, ISO/IEC TR 18015:2006(E).
  - Includes information on C's <iohw.h> and C++'s <hardware>.
- "The Embedded C Extension to C," Marcel Beemster *et al.*, *C/C++ Users Journal*, August (Part 1) and September (Part 2), 2005.
  - Discusses <iohw.h>.
    - Purely a C approach – no C++.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

Compilers and volatile:

- "Volatiles are Miscompiled, and What to Do about It," Eric Eide and John Regehr, *Proc. Eighth ACM and IEEE Intl. Conf. on Embedded Software (EMSOFT)*, October 2008.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

Implementing callbacks from C:

- "Interrupts in C++," Alan Dorfmeyer and Pat Baird, *Embedded Systems Programming*, August 2001.
- "Applying Design Patterns to Simplify Signal Handling," Douglas C. Schmidt, *C++ Report*, April 1998.
- "Use Member Functions for C-Style Callbacks and Threads – a General Solution," Daniel Lohmann, *The Code Project*, 8 July 2001, http://www.codeproject.com/win32/callback_adapter.asp.
- "Generalizing C-Style Callbacks," http://www.crystalclearsoftware.com/cgi-bin/boost_wiki/wiki.pl?Generalizing_C-Style_Callbacks.
- "Serial Port Design Pattern," http://www.eventhelix.com/RealtimeMantra/PatternCatalog/serial_port_design_pattern.htm.
- "Encapsulating ISRs in C++," Daniel G. Rusch, *Embedded Systems Programming*, February 1998.
- "Interoperability & C++ Compilers," Joe Goodman, *C/C++ Users Journal*, March 2004.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

ISR issues:

- "Reduce RTOS latency in interrupt-intensive apps," Nick Lethaby, *Embedded Systems Design*, June 2009.
- "Minimize Your ISR Overhead," Nigel Jones, *Embedded Systems Design*, January 2007.
  - ➧ Primarily about avoiding unnecessary register saves/restores during ISR invocations.
- "Modeling Interrupt Vectors," Dan Saks, *Embedded Systems Design*, September 2006.
  - ➧ Focuses on getting ISR addresses into interrupt vector tables.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

TR1 and Boost:

- *The C++ Standard Library Extensions*, Pete Becker, Addison-Wesley, 2007, ISBN 0-321-41299-0.
  ➧ A comprehensive reference for TR1.
- Scott Meyers' TR1 Information web page, http://www.aristeia.com/EC3E/TR1_info.html.
  ➧ Contains links to proposal documents, articles, books, etc.
- *Effective C++, Third Edition*, Scott Meyers, Addison-Wesley, 2005.
  ➧ Item 35 explains and demonstrates use of tr1::function.
  ➧ The TOC is attached.
- "Generalized Function Pointers," Herb Sutter, *C/C++ Users Journal Experts Forum*, August 2003.
  ➧ Describes std::tr1::function.
- Boost web site, http://www.boost.org/
- *Beyond the C++ Standard Library: An Introduction to Boost*, Björn Karlsson, Addison-Wesley, 2006, ISBN 0-321-13354-4.
  ➧ An overview of selected Boost libraries, including bind and function.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

Compile-time dimensional unit analysis:

- "Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation," Walter E. Brown, Second Workshop on C++ Template Programming, October 2001. Available at http://www.oonumerics.org/tmpw01/brown.pdf.
- "Dimension Checking of Physical Quantities," Michael Kenniston, *C/C++ Users Journal*, November 2002.
  ➧ A description of an implementation for less conformant compilers (e.g., Visual C++ 6).
- *Boost.Units*, Matthias C. Schabel and Steven Watanabe, http://www.boost.org/doc/libs/1_46_0/doc/html/boost_units.html.
- "Library for checking dimensional unit correctness?," comp.lang.c++.moderated thread, initiated 22 October 2005. Available at http://tinyurl.com/yrgwt5.
- "Interest in dimension/unit-checking library?," Boost mailing list thread, initiated 19 October 2005. Available at http://tinyurl.com/cwveo.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

Implementing FSMs:

- *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, David Abrahams and Aleksey Gurtovoy, Addison-Wesley, 2005, ISBN 0-321-22725-5, Chapter 11.
  - ➧ Code for FSM example also available at http://boost.org/libs/mpl/example/fsm/player1.cpp.
  - ➧ Source code used per the Boost Software License, Version 1.0:

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



## Further Information

More on implementing FSMs:

- "UML Tutorial: Finite State Machines," Robert C. Martin, *C++ Report*, June 1998.
- "Yet Another Hierarchical State Machine," Stefan Heinzmann, *Overload*, December 2004.
- "Hierarchical State Machine Design in C++," Dmitry Babitsky, *C/C++ Users Journal*, December 2005.
- "State Machine Design Pattern," Anatoly Shalyto *et al.*, *Proceedings of .NET Technologies 2006*, May 2006.
- "State Machine Design in C++," David Lafreniere, *C/C++ Users Journal*, May 2000.
- "The Anthology of the Finite State Machine Design Patterns," Paul Adamczyk, *Proceedings of PLoP 2003*, September 2003.
  - ➧ A summary of 24 FSM design patterns!
- "The Boost Statechart Library," Andreas Huber Dönni, April 2007, http://www.boost.org/doc/libs/1_36_0/libs/statechart/doc/index.html.




© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

The Curiously Recurring Template Pattern (CRTP):

- "Curiously Recurring Template Patterns," James O. Coplien, *C++ Report*, February 1995.
- Many template-oriented C++ books include a discussion of CRTP.

Template metaprogramming:

- *Modern C++ Design: Generic Programming and Design Patterns Applied*, Andrei Alexandrescu, Addison-Wesley, 2001, ISBN 0-201-70431-5.
- *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, David Abrahams and Aleksey Gurtovoy, Addison-Wesley, 2005, ISBN 0-321-22725-5.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

Guidelines for using C++ in safety-critical software:

- *Guidelines for the Use of the C++ Language in Critical Systems*, MISRA, June 2008, ISBN 978-906400-03-3 (hardcopy) or 978-906400-04-0 (PDF).
- *High-Integrity C++ Coding Standard Manual (Version 2.4)*, Programming Research, December 2006.
- "The Power of 10: Rules for Developing Safety-Critical Code," Gerard J. Holzmann, *IEEE Computer*, June 2006.
- *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, Lockheed Martin Corporation, December 2005.
- *Cert C++ Coding Standard*, CERT, http://tinyurl.com/za3hr.
  ➧ Focuses on developing secure code.
- *Guidelines for the Use of the C Language in Critical Systems*, MISRA, October 2004, ISBN 0 9524156 2 3 (hardcopy) or 0 9524156 4 X (PDF).
  ➧ Superceded for C++ development by the C++ version (above).


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

Aspects of C++ with unpredictable (e.g., undefined) behavior:

- *An Investigation of the Unpredicatable Features of the C++ Language*, M. G. Hill and E. V. Whiting, QinetiQ Ltd., May 2004.

Static analysis for safety-critical software:

- "Using Static Analysis to Evaluate Software in Medical Devices," Raoul Jetley and Paul Anderson, *Embedded Systems Design*, April 2008.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Further Information

C++ in real-time systems:

- "Worst Case Execution Time," Wikipedia, http://en.wikipedia.org/wiki/Worst_case_execution_time.
- "Use of Modern Processors in Safety-Critical Applications," Iain Bate *et al.*, *The Computer Journal*, June 2001.
  - ➧ Section 4.4. is devoted to WCET analysis.
- "You Can't Control what you Can't Measure...," Nat Hillary and Ken Madsen, *Proceedings of the 2nd Intl. Workshop on Worst Case Execution Time Analysis*, June 2002.
  - ➧ Overview of pros/cons of some approaches to analyzing RT software.
    - ◆ Written by marketing managers – and it shows.


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

## Further Information

C++11:

- "C++11 – auch ein Stimmungsbild," Marc Mutz, *heise Developer*, 21 September 2011.
  ➧ Schliesst einen Überblick der "Top 11" neuen Features ein.
- *C++0x*, Wikipedia, http://en.wikipedia.org/wiki/C%2B%2B0x.
- *Overview of the New C++ (C++0x)*, Scott Meyers, http://www.artima.com/shop/overview_of_the_new_cpp.
  ➧ Annotated training materials (analogous to these).
- *C++0x: A quick and dirty introduction*, Olve Maudal and Lars Gullik Bjønnes, November 2007, http://www.pvv.org/~oma/cpp0x_aquadi_nov_2007.pdf.
- *Working Draft, Standard for Programming Language C++*, Pete Becker (Ed.), 2010-11-27, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3225.pdf.
  ➧ Final freely available draft.

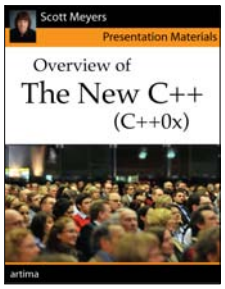

Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Lizenzinformation

Scott Meyers lizenziert die Unterlagen für diesen und andere Schulungskurse zu kommerziellen oder persönlichen Zwecken. Details:

- Kommerzielle Zwecke: http://aristeia.com/Licensing/licensing.html
- Persönliche Nutzung: http://aristeia.com/Licensing/personalUse.html

Verfügbare Kurse für persönliche Nutzung sind:






Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.



© 2011 Scott Meyers, alle Rechte vorbehalten.

# Über Scott Meyers

Scott ist Trainer und Fachberater für den Entwurf und die Implementierung von C++ Softwaresystemen.

Auf seiner Website

http://www.aristeia.com/

finden Sie folgende Informationen:

- Schulung und Fachberatung
- Bücher, Artikel und andere Veröffentlichungen
- Bevorstehende Präsentationen
- Blog über berufliche Tätigkeiten


Urheberrechtlich geschütztes Material, alle Rechte vorbehalten.


## Inhalt







Übernommen von Scott Meyers: *Effektiv C++ programmieren: 55 Möglichkeiten, Ihre Programme und Entwürfe zu verbessern*, Addison-Wesley, 2006.


© 2011 Scott Meyers, alle Rechte vorbehalten.

Übernommen von Scott Meyers: *Effektiv C++ programmieren: 55 Möglichkeiten, Ihre Programme und Entwürfe zu verbessern*, Addison-Wesley, 2006.

# Inhaltsverzeichnis



Übernommen von Scott Meyers: *Mehr Effektiv C++ programmieren: 35 neue Wege zur Verbesserung Ihre Programme und Entwürfe*, Addison-Wesley, 1997.



© 2011 Scott Meyers, alle Rechte vorbehalten.

Übernommen von Scott Meyers: *Mehr Effektiv C++ programmieren: 35 neue Wege zur Verbesserung Ihre Programme und Entwürfe,* Addison-Wesley, 1997.

## Contents



Reprinted from *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library,* by Scott Meyers, Addison-Wesley Publishing Company, 2001.

Scott Meyers, Fachberater für Softwareentwicklung
http://www.aristeia.com/

© 2011 Scott Meyers, alle Rechte vorbehalten.

Reprinted from *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, by Scott Meyers, Addison-Wesley Publishing Company, 2001.



© 2011 Scott Meyers, alle Rechte vorbehalten.