C++ IN DER EMBEDDED-ENTWICKLUNG – WAS DRAN IST AN DEN VORURTEILEN

Günter Obiltschnig
Applied Informatics Software Engineering GmbH
St. Peter 33
9184 St. Jakob im Rosental
Austria
guenter.obiltschnig@appinf.com

Ein Weg um die steigende Komplexität in der Embedded-Software zu bändigen ist der Einsatz einer objektorientierten Programmiersprache. Dabei erfreut sich insbesondere C++ zunehmender Beliebtheit. Trotzdem gibt es, was den Einsatz von C++ in Embedded-Systemen betrifft, immer noch viele Vorurteile und Vorbehalte. Auf einige davon wird in diesem Text eingegangen, indem Code-Beispiele und der daraus vom Compiler erzeugte Code analysiert werden.

Einleitung

In der Entwicklung von Embedded-Systemen sieht man sich zunehmend mit dem Problem rapide steigender Anforderungen, und somit steigender Komplexität der resultierenden Software konfrontiert. Ein Weg diese Komplexität zu bändigen ist der Einsatz einer objektorientierten Programmiersprache. Dabei erfreut sich insbesondere C++ zunehmender Beliebtheit. Durch den Einsatz im Joint Strike Fighter Projekt [1] ist C++ auch im Bereich sicherheitskritischer Systeme wie z. B. Avionik salonfähig geworden. Trotzdem gibt es, was den Einsatz von C++ in Embedded-Systemen betrifft, immer noch viele Vorurteile und Vorbehalte. Beispielsweise sei C++ zu langsam, bzw. zu ressourcenintensiv für den Einsatz auf Embedded-Systemen, oder die Programmiersprache wird schlicht als zu komplex und somit schwer beherrschbar empfunden.

Die Performance eines C++ Programms

Beim Entwurf der Programmiersprache C++ wurde großer Wert auf die Rückwärtskompatibilität zu C gelegt. Diese betrifft nicht nur den Sprachumfang, sondern auch das Laufzeitverhalten compilierter Programme. Das Laufzeitverhalten eines C Programms ändert sich nicht, wenn es mit einem C++ Compiler übersetzt wird. Eine grundlegende Eigenschaft von C++ ist, dass Sprachfeatures, welche in einem Programm nicht verwendet werden, auch keinen Overhead verursachen („you don't pay for what you don't use") [2].

Klassen und Methoden

Klassen in C++ sind zunächst einmal vergleichbar mit Strukturen in C. Solange keine virtuellen Elementfunktionen oder Vererbung benutzt werden, hat eine C++ Klasse die exakt gleiche Abbildung im Hauptspeicher wie eine äquivalente C Struktur. Die Zugriffskontrolle auf Elemente mit Hilfe der Keywords *public*, *protected* und *private* hat keine Auswirkung auf die Laufzeit. Elementfunktionen einer Klasse, sofern sie nicht als *virtual* deklariert sind, wirken sich nicht auf den Speicherbedarf einer Klasse aus. Der Aufruf einer (nicht-*virtual* und nicht-*static*) Elementfunktion hat den selben Aufwand wie der Aufruf einer C Funktion, mit dem einzigen Unterschied, dass ein zusätzlicher Parameter – der *this* Pointer – übergeben wird. Beispielsweise erzeugt der GNU C++ Compiler für den C++ Code in Beispiel 1 den exakt gleichen Object-Code wie der GNU C Compiler für den Code in Beispiel 2. Dies kann leicht verifiziert werden, indem der GNU C, bzw. C++ Compiler mit den Optionen *–Wa,-ahlsdn -gstabs* aufgerufen wird. Die beiden Optionen veranlassen den Compiler ein Assembler-Listing zu produzieren, welches mit dem ursprünglichen C, bzw. C++ Code annotiert ist.

```
#include <cstdio>

class Complex
{
public:
      Complex();
      void print() const;

private:
      double _real;
      double _imag;
};

Complex::Complex():
      _real(0.0),
      _imag(0.0)
{
}

void Complex::print() const
{
      std::printf("%d+%di", _real, _imag);
}
```

Beispiel 1: Einfache C++ Klasse

Anzumerken ist, dass der Code in Beispiel 1 ein schlechtes Beispiel für C++ Design ist. Es fehlen unter anderem weitere Konstruktoren und Destruktor, sowie Zuweisungsoperator, welche der Einfachheit halber weggelassen wurden.

```
#include <stdio.h>

struct Complex
{
      double _real;
      double _imag;
};

void Complex_init(struct Complex* pComplex)
{
      pComplex->_real = 0.0;
      pComplex->_imag = 0.0;
}

void Complex_print(const struct Complex* pComplex)
{
      printf("%d+%di", pComplex->_real, pComplex->_imag);
}
```

Beispiel 2: Der äquivalente C Code zum C++ Code in Beispiel 1

Konstruktoren und Destruktoren

Wird in einem C++ Programm eine Instanz einer Klasse z. B. auf dem Stack angelegt, wird automatisch der Konstruktor dieser Klasse aufgerufen, um das Objekt ordnungsgemäß zu initialisieren. Daher entspricht der folgenden C++ Deklaration:

```
Complex myComplex;
```

folgender C Code:

```
struct Complex myComplex;
Complex_init(&myComplex);
```

C++ Destruktoren haben kein entsprechendes C Konstrukt. Der Grund dafür ist, dass der C++ Compiler garantiert, dass für jedes Objekt, welches auf dem Stack erfolgreich angelegt wurde, der Destruktor aufgerufen wird, sobald der Block in dem das Object angelegt wurde, verlassen wird, egal auf welche Art (*return*, *break*, *goto*, Exception). Dies ist eine der größten Stärken von C++, ermöglicht es doch das RAII-Idiom (Resource Acquisition is Initialization) [3], welches einen großen Beitrag zur Vermeidung von Memory-Leaks und anderen Programmierfehlern in C++ Programmen leistet.

Der automatische Aufruf von Konstruktoren und Destruktoren kann einen starken Einfluss auf das Laufzeitverhalten haben. Der reine Overhead für den Aufruf eines Konstruktors, bzw. Destruktors ist nicht größer als jener für einen Funktionsaufruf in C (bzw. im Falle von als *virtual* deklarierten Destruktoren, ein indirekter Funktionsaufruf über einen Funktionspointer). Allerdings ist es für den Benutzer einer Klasse nicht immer offensichtlich, was in einem Konstruktor und Destruktor alles passiert (z. B. Allokation von Speicher am Heap). Zu häufiges Erzeugen von (temporären) Objekten am Stack (z. B. bei Verwendung von Klassen als Funktionsparameter oder Rückgabewerten) und die damit verbundenen Ausführungen von Konstruktoren und Destruktoren sind in der Tat eine häufige Ursache für langsame C++ Programme, speziell bei unerfahrenen C++ Programmierern. Das Programm in Beispiel 3 zeigt, was zu vermeiden ist.

```
#include <string>

class Name
{
public:
      void setFirst(std::string first);
      std::string getFirst() const;

      void setLast(std::string last);
      std::string getLast() const;

private:
      std::string _first;
      std::string _last;
};

void Name::setFirst(std::string first)
{
      _first = first;
}

std::string Name::getFirst() const
{
      return _first;
}

...
```

Beispiel 3: Unnötiges Kopieren von Objekten

Bei jedem Aufruf von *setFirst()* muss das übergebene *std::string* Objekt kopiert werden, was unter anderem eine Speicherallokation auf dem Heap, sowie ein Kopieren des String-Inhaltes zur Folge hat. Das gleiche trifft auf *getFirst()* zu. Wie es besser geht, zeigt Beispiel 4.

```
#include <string>

class Name
{
public:
      void setFirst(const std::string& first);
      const std::string& getFirst() const;

      void setLast(const std::string& last);
      const std::string& getLast() const;
```

```
private:
      std::string _first;
      std::string _last;
};

inline void Name::setFirst(const std::string& first)
{
      _first = first;
}

inline const std::string& Name::getFirst() const
{
      return _first;
}

...
```

Beispiel 4: Effizientes C++

In Beispiel 4 werden Referenzen verwendet, um das unnötige Kopieren von Objekten zu vermeiden. Die Angabe von *inline* veranlasst den Compiler, anstelle eines Funktionsaufrufs direkt den Code der Funktion, soweit möglich, an die Stelle des Aufrufs zu kopieren (vergleichbar mit Macros in C). Ein Aufruf von

```
name.setFirst(firstName);
```

wird, unter Mithilfe des Optimizers im erzeugten Object Code zum Äquivalent von

```
name._first = firstName;
```

Das *inline* Keyword, Referenzen, sowie auch Templates sind die besten Hilfsmittel, um hochperformanten C++ Code zu schreiben.

Vererbung und Polymorphie

Objekt-orientierte Programmierung lebt unter anderem von Vererbung und Polymorphie, bzw. dynamischer Bindung. Auch wenn der Trend in C++ mehr in Richtung generischer Programmierung und weg von „klassischer“ objekt-orientierter Programmierung geht, spielen Vererbung und Polymorphie eine wichtige Rolle in C++. Dementsprechend lohnt sich ein Blick auf den damit verbundenen Overhead. Reine einfache Vererbung bringt zunächst einmal keinerlei Overhead mit sich – weder in Bezug auf Laufzeit noch in Bezug auf den Speicherverbrauch. Gezeigt werden kann dies durch den C++ Code in Beispiel 5, und dem äquivalenten C Code in Beispiel 6.

```
class Base
{
    // ...
};

class Derived: public Base
{
    // ...
};
```

Beispiel 5: Vererbung in C++

```
struct Base
{
    // ...
};

struct Derived
{
    struct Base base;
    // ...
```

```
};
```

Beispiel 6: Vererbung nachgebildet in C

Bei einfacher Vererbung werden in der abgeleiteten Klasse die Elemente der Basisklasse einfach vor die Elemente der abgeleiteten Klasse gelegt. Beim Aufruf einer Elementfunktion entsteht kein zusätzlicher Overhead. Etwas anders sieht es bei mehrfacher Vererbung aus. Hier entsteht ein geringer Overhead (weniger als 25 %) beim Aufruf von Elementfunktionen dadurch dass eventuell der *this*-Pointer angepasst werden muss.

Untrennbar verbunden mit Vererbung ist die dynamische Bindung, bzw. Polymorphie. Kennzeichnend für das Prinzip von C++ unnötigen Overhead zu vermeiden ist, dass polymorphe Funktionen mit dem *virtual* Keyword explizit deklariert werden müssen. Das Hinzufügen von *virtual* Elementfunktionen zu einer Klasse hat zwei Auswirkungen. Erstens wird die Klasse um ein zusätzliches, verstecktes Datenelement, die sogenannte Virtual Function Table erweitert. Zweitens werden Funktionsaufrufe auf eine *virtual* Funktion indirekt, über die Virtual Function Table, durchgeführt. Dies lässt sich am besten über einen Vergleich von C++ und äquivalentem C Code durchführen. Beispiel 7 zeigt eine C++ Klasse mit einer virtuellen Elementfunktion.

```
class Dynamic
{
public:
    virtual void doSomething();
};

void Dynamic::doSomething()
{
    // ...
}
```

Beispiel 7: C++ Klasse

Beispiel 8 zeigt den zum C++ Code in Beispiel 7 äquivalenten C Code.

```
struct Dynamic_VTBL;
struct Dynamic
{
    struct Dynamic_VTBL* _pVtbl;
};

struct Dynamic_VTBL
{
    // ...
    void (*doSomething)(struct Dynamic* this);
};

void Dynamic_doSomething(struct Dynamic* this)
{
}
```

Beispiel 8: Implementierung einer Virtual Function Table in C

Jeder Aufruf der Elementfunktion doSomething() muss nun über die Virtual Function Table erfolgen. Demnach entspricht folgendem C++ Code:

```
pDynamic->doSomething();
```

folgender C Code:

```
(*pDynamic->_pVtbl->doSomething)(pDynamic);
```

Der Pointer auf die Virtual Function Table eines Objektes muss initialisiert werden. Der Compiler generiert automatisch den Code dafür in den Konstruktoren der Klasse. Wird vom Programmierer kein Konstruktor definiert erzeugt der Compiler selbst einen. Pro Klasse gibt es im Speicher eine Instanz der dazugehörenden Virtual Function Table welche dann von allen Objekten dieser Klasse benutzt wird. Auf entsprechenden Systemen kann die Virtual Function Table im ROM abgelegt werden.

Neben den Funktionspointern kann die Virtual Function Table noch weitere Informationen enthalten, beispielsweise Informationen zur Mehrfachvererbung oder Laufzeit-Typinformation (RTTI).

Typ-Konvertierungen

C++ bietet neue Operatoren zur Konvertierung von Datentypen – *reinterpret_cast*, *static_cast* und *dynamic_cast*. Die ersten beiden, *reinterpret_cast* und *static_cast* entsprechen exakt dem Cast-Operator in C. Nicht so *dynamic_cast* – dieser ist mit einem Laufzeit-Overhead verbunden, da die Gültigkeit des Casts zur Laufzeit getestet werden muss. Dies erfolgt über eine Abfrage der Laufzeit-Typinformation (RTTI). Der Aufwand hierfür ist Abhängig von der konkreten Implementierung des Compilers und kann Abhängig von der Vererbungstiefe sein.

Namespaces

Die Verwendung von Namespaces wirkt sich nicht auf die Laufzeit des Programmes aus. Namespaces wirken sich alleine auf die Symboltabelle eines Programmes aus. Dies kann bei dynamisch gelinkten Programmen die Startup-Zeit (vernachlässigbar) beeinflussen.

Exceptions

Exceptions sind das C++ Feature mit der größten Auswirkung, sowohl auf die Ausführungszeit, als auch den Speicherbedarf eines Programms. Wie sich die Verwendung von Exceptions auswirkt, hängt stark vom verwendeten Compiler ab. Moderne Compiler erzeugen Code, der für den Fall, dass keine Exception auftritt, auch keinen Overhead in der Ausführungszeit bewirkt. Der Overhead beschränkt sich hier auf zusätzliche statische Daten, und zusätzlichen Code im Laufzeitsystem des Compilers (ca. 15 %). Man sollte hier allerdings nicht in die Falle tappen und den Overhead für Exceptions mit der Fehlerbehandlungsstrategie „alle Fehler ignorieren“ vergleichen. Auch herkömmliche Fehlerbehandlungsstrategien verursachen Overhead, insbesonders auch in der Entwicklungszeit. All zu oft wird auf korrekte Fehlerbehandlung verzichtet, was sich fatal auf die Zuverlässigkeit eines Systems auswirken kann. Viele Fehlerbehandlungsmethoden wirken sich auch negativ auf die Lesbarkeit und Wartbarkeit eines Programms aus. Exceptions stellen hier eine wirklich gute Alternative dar – sie machen sowohl den Code les- und wartbarer, als auch das Programm robuster. Bei strengen Echtzeit-Anforderungen ist allerdings Vorsicht geboten – die zum Abarbeiten einer Exception benötigte Zeit hängt einerseits vom Compiler, andererseits davon ab, wo im Programm eine Exception auftritt, und wo sie abgefangen wird. Eine Berechnung dieser Zeit ist nur mit hohem Aufwand und genauer Kenntnis des Compilers möglich.

Templates

Templates sind das leistungsfähigste Konstrukt in C++, aber auch jenes Konstrukt mit dem größten Potential dazu, die Codegröße explodieren zu lassen. Richtig eingesetzt, lässt sich mit Hilfe von Templates hochperformanter und sicherer Code schreiben. Beispielsweise erzeugt der Compiler (GCC 4.0.3 für ARM) aus dem Programm in Beispiel 9 den in Beispiel 10 gezeigten Assembler Code.

```
template <typename C>
C add(C a, C b)
{
      return a + b;
}

int main(int argc, char** argv)
{
      int x = add(2, 3);

      return x;
}
```

Beispiel 9: Verwendung eines Funktionstemplates

```
   1                    .file   "add.cpp"
```

```
 2                      .text
 3                      .align  2
 4                      .global main
 6              main:
 7                      @ args = 0, pretend = 0, frame = 0
 8                      @ frame_needed = 0, uses_anonymous_args = 0
 9                      @ link register save eliminated.
10 0000 0500A0E3        mov     r0, #5
11                      @ lr needed for prologue
12 0004 0EF0A0E         mov     pc, lr
```

Beispiel 10: Optimierter Assembler Code

Die Standardbibliothek

Ein wichtiger Bestandteil von C++ ist die Standardbibliothek, welche häufig benötigte Datenstrukturen und Algorithmen in generischer Form bereitstellt. Für kleine Embedded-Systeme ist diese allerdings nicht geeignet. Beispielsweise lässt die Verwendung einiger weniger Elementfunktionen von *std::vector* die Größe eines statisch gelinkten Executables schon um etwa 80 KByte wachsen. Hat man allerdings 32 MByte oder mehr an RAM zur Verfügung lohnt es sich schon nicht mehr über die Verwendung der Standardbibliothek nachzudenken – man verwendet sie einfach.

Die Komplexität von C++

C++ ist eine Programmiersprache, die einerseits fast unbegrenzte Möglichkeiten bietet, andererseits aber auch einen entsprechenden Lernaufwand voraussetzt. Sicherlich ist der Aufwand um C++ einigermaßen zu beherrschen um ein Vielfaches größer als bei C. Man wird dafür aber mit einer höheren Produktivität und leichter wartbarem Code belohnt. Man muss keineswegs ein C++ Profi sein, um von C++ zu profitieren – solide Kenntnisse der objekt-orientierten Programmierung sollten allerdings vorhanden sein. Ob man sich schlussendlich auch mit schwierigen Themen wie Template-Metaprogrammierung beschäftigt [4], sei jedem selbst überlassen.

Zusammenfassung

Viele der Vorurteile gegenüber C++ – besonders jene die Performance betreffenden – erweisen sich dank moderner Compiler als nicht (mehr) berechtigt. Dank einiger Eigenschaften von C++ ist es einem guten Compiler sogar möglich, schnelleren und sichereren Code zu erzeugen als ein C Compiler. Die höhere Komplexität von C++ stellt allerdings ein nicht unerhebliches Hindernis für den schnellen Umstieg von C auf C++ dar. Die Möglichkeit, auf einer viel höheren Abstraktionsebene wesentlich produktiver programmieren zu können, ohne dabei negative Auswirkungen auf die Laufzeit zu haben, ist es jedoch, was C++ zu einer hervorragenden Programmiersprache für die Entwicklung von Embedded-Systemen macht.

Quellen

[1] Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, Lockheed Martin Corporation, 2005.
http://www.research.att.com/~bs/JSF-AV-rules.pdf

[2] Technical Report on C++ Performance, ISO/IEC PDTR 18015, 2003.
http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf

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

[4] Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley, 2001.

Zum Autor

Günter Obiltschnig ist Geschäftsführer der Applied Informatics Software Engineering GmbH und Gründer des POCO C++ Libraries Open Source Projektes. Er verfügt über 15 Jahre Erfahrung in der Entwicklung von Software für verschiedene Systeme, von verteilten Unternehmensapplikationen bis hin zu Microcontroller-basierten Geräten. Seit mehreren Jahren beschäftigt er sich intensiv mit dem Einsatz von C++ für Embedded-Systeme.