C++ für sicherheitskritische Systeme

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

Einleitung

C++ [1] wird immer häufiger für Embedded Software eingesetzt – auch für sicherheitskritische und harte Echtzeitsysteme. Aufgrund seines Designs ist C++ jedoch nicht optimal für sicherheitskritische Systeme geeignet. C++ gibt dem Entwickler viele Freiheiten, und verlangt somit ein hohes Maß an Verantwortung. Trotzdem gibt es gute Gründe, die für den Einsatz von C++ in sicherheitskritischen Systemen sprechen:

- Höhere Abstraktionsebene. C++ ermöglicht die Arbeit auf einer wesentlich höheren Abstraktionsebene als C, ohne jedoch auf die Möglichkeit der hardwarenahen Programmierung verzichten zu müssen.
- Objektorientierte Programmierung. C++ bietet direkte Unterstützung für objektorientierte Programmierung.
- Portabilität. Die mittlerweile gute Verfügbarkeit von standardkonformen C++ Compilern ermöglicht zu jeder Zeit eine einfache Portierung einer Applikation auf eine neue, bessere Prozessorplattform.
- Effizienz. Gute C++ Compiler erzeugen mindestens genau so effizienten Code wie gute C Compiler. In einigen Fällen kann sogar noch effizienterer Code erzeugt werden.
- Verfügbarkeit von Entwicklern. C++ ist eine der am häufigsten verwendeten Programmiersprachen – somit steht eine große Anzahl von C++ Entwicklern zur Verfügung.
- Werkzeugunterstützung. Viele Werkzeuge der modellbasierten Softwareentwicklung können C++ Code generieren.
- Ausgereiftheit. C++ ist eine ausgereifte Programmiersprache, die sich im Praxiseinsatz vielfach bewährt hat.

Andererseits hat C++ natürlich auch viele Eigenschaften, welche die Eignung für sicherheitskritische Systeme einschränken:

- C++ ist eine hoch komplexe Sprache. Das Erlernen und Beherrschen der Sprache erfordert einen hohen Lernaufwand. C++ Code macht nicht immer das, was man “intuitive” beim Lesen des Codes erwarten würde. Die hohe Komplexität erhöht auch die Wahrscheinlichkeit von Compilerfehlern – auch Compiler-Programmierer sind nur Menschen.
- Die Semantik einiger C++ Konstrukte ist nicht vollständig spezifiziert. Dies lässt Raum für Portabilitätsprobleme – das Verhalten eines Programms kann Abhängig davon sein welcher Compiler benutzt wurde (undefiniertes, unspezifiziertes und implementierungsabhängiges Verhalten).

- C++ macht es einfach Fehler zu machen die vom Compiler nicht immer diagnostiziert werden können (Beispiel: Zuweisungsoperator '=' und Vergleichsoperator '==').
- C++ hat keine automatischen Laufzeitprüfungen (z. B. Overflow, Array-Indizierung).
- Obwohl C++ eine streng typisierte Sprache ist gibt es zu viele Löcher im Typsystem über die es, absichtlich oder unabsichtlich, übergangen werden kann.

Beim Einsatz von C++ für sicherheitskritische Systeme muss sorgfältig darauf geachtet werden Sprachkonstrukte und Code welche zu unerwünschtem Programmverhalten führen können zu vermeiden. Es sei darauf hingewiesen, dass auch C fast alle dieser Probleme, und noch einige mehr, hat. Dennoch ist C eine der am meisten verwendeten Sprachen für sicherheitskritische Systeme. Ermöglicht wird dies durch Codierungsstandards, welche die erlaubten Sprachkonstrukte auf eine "sichere" Teilmenge beschränken.

Codierungsstandards

Der populärste Codierungsstandard für C in sicherheitskritischen Systemen ist MISRA-C: Guidelines for the use of the C language in critical systems [2]. MISRA-C spezifiziert eine "sichere" Teilmenge von C in Form von 121 verpflichtenden und 20 empfohlenen Regeln. Seit der Erstveröffentlichung 1998 (und einer Überarbeitung 2004) wird MISRA-C sehr oft verwendet.

Der "Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program" [3], kurz "JSF C++" war fast revolutionär, hat er doch eine Abkehr von Ada als alleinige Programmiersprache für Avioniksoftware bei Projekten des US Verteidigungsministeriums signalisiert. Obwohl JSF C++ auf vielen MISRA-C Regeln aufbaut, ist der Codierungsstandard doch konzeptuell sehr unterschiedlich im Vergleich zu MISRA-C (oder auch dem neuen MISRA-C++). So definiert JSF C++ z. B. auch Codestil und Codemetrik Richtlinien, welche es bei MISRA nicht gibt. Beispiel: Regel 1: *"[a]ny one function (or method) will contain no more than 200 logical source lines of code*", oder Regel 50: *"[t]he first word of the name of a class, structure, namespace, enumeration, or type created with typedef will begin with an uppercase letter. All others letters will be lowercase.*" Insgesamt definiert JSF C++ 221 Regeln.

Der Trend hin zu C++ hat im Sommer 2008 zur Veröffentlichung von MISRA-C++ [4] geführt. Basierend auf MISRA-C definiert der neue Standard 228 Regeln für den Einsatz von C++ in sicherheitskritischen Systemen.

Probleme mit C++ in sicherheitskritischen Systemen

Eine Analyse von MISRA-C++ und JSF C++ bringt viele potentielle Probleme ans Tageslicht, welche beim Einsatz von C++ in sicherheitskritischen Systemen beachtet werden müssen.

Präprozessor

Der Einsatz der Präprozessors wird durch die Codierungsstandards stark eingeschränkt und im Wesentlichen auf "include guards" welche das mehrfache Inkludieren des selben Headers verhindern, reduziert. Die Verwendung von Makros zur Definition von Konstanten oder als Funktionsmakros ist verboten. C++ Sprachelemente, wie z. B. das *const* Schlüsselwort, Aufzählungstypen oder *inline* Funktionen sind ein mehr als ausreichender Ersatz für Makros.

Implementierungsabhängiges, unspezifiziertes und undefiniertes Verhalten

Der C++ Standard spezifiziert das exakte Verhalten der meisten, aber nicht aller Sprachelemente. Bestimmte Sprachelemente werden vom Standard als implementierungsabhängig ("implementation-defined"), unspezifiziert ("unspecified") oder undefiniert ("undefied") beschrieben.

Implementierungsabhängig bedeutet, dass der Compiler-Programmierer weitgehend frei ist, ein bestimmtes Sprachkonstrukt so zu implementieren wie er es wünscht, vorausgesetzt dass das implementierte Verhalten konsistent und dokumentiert ist, und das die Compilierung von Standard-konformem Code erfolgreich ist. Beispiel: Speichergröße von Integer-Typen.

Ähnlich ist es mit unspezifiziertem Verhalten – hier muss das Verhalten der Implementierung aber weder konsistent, noch dokumentiert sein. Beispiel: Evaluierungsreihenfolge von Funktionsargumenten.

Bei undefiniertem Verhalten gibt es keinerlei Vorgaben – bereits die Compilierung eines undefinierten Sprachkonstrukts kann fehlschlagen, oder das laufende Programm kann falsche Resultate liefern. Beispiel: Dereferenzierung eines Null-Pointers.

Code welcher auf implementierungsabhängigem, unspezifiziertem, oder undefiniertem Verhalten basiert ist nicht portabel und somit verboten. Ebenso untersagt ist die Verwendung von Compiler-spezifischen Spracherweiterungen.

Fehleranfällige Sprachkonstrukte

C++ hat wie C einige Sprachkonstrukte die Fehler geradezu herausfordern, wie z. B. *switch* (vergessener Fall, vergessenes *break*). Codierungsstandards setzen hier an und schränken die Verwendung dieser Konstrukte ein (z. B. wo *case* plaziert werden darf, verlangtes *break* oder *throw* am Ende jeder *case* Klausel, *default* Fall).

Bei Kontrollstrukturen wie *if*, *else*, *switch*, *while*, *for* und *do ... while* muss ein Block verwendet werden (Ausnahme: *else if*). Jedes *if ... else* muss mit einem *else* beendet werden. Bei *for* Schleifen gibt es Regelungen für Zähler- und Kontrollvariablen. *Goto* darf nur sehr eingeschränkt verwendet werden; *setjmp/longjmp* ist verboten.

Typsystem

Das C++ Typsystem hat viele Löcher, die eine Umgehung einfach machen. Codierungsstandards schränken die Verwendung von impliziten oder expliziten Typkonvertierungen (casts) stark ein (möglicher Informationsverlust). C++ definiert keine fixen Speichergrößen für die eingebauten Typen. Dies macht es schwierig, portablen Code welcher z. B. Integer-Variablen fixer Größe benötigt, zu schreiben. C++, wie auch C, führt alle arithmetischen Operationen als *int* oder *long* aus, abhängig vom Operandentyp. Vor einer arithmetischen Operation werden integrale Typen (*signed*/*unsigned short* und *signed*/*unsigned char*) nach *int*/*unsigned int* gewandelt. Dies kann zu nicht-portablen Code führen, wie im folgenden Beispiel gezeigt:

```
short i = 10000;
short j = 8;
int32 result = i * j;
```

Auf einem System mit 32 bit *int* (und 16 bit *short*) ist *result* wie erwartet 80000. Auf einem System mit 16 bit *int* ist das Ergebnis jedoch undefiniert, da die Multiplikation in 16-bit *int* ausgeführt wird, ein Überlauf auftritt, und das somit falsche Ergebnis nach *int32* konvertiert wird.

Der einfache *char* Typ (ohne *signed*/*unsigned* Präfix) ist problematisch, da er entweder *signed* oder *unsigned* sein kann. Codierungsstandards erlauben den einfachen *char* Typ nur zur Speicherung von Zeichen. Arithmetische Operationen mit *char* sind verboten.

Klassen

MISRA-C++ und JSF C++ haben einen starken Fokus auf Datenkapselung. Das heißt das Instanzvariablen einer Klasse generell *private* sein müssen. Jegliche Umgehung der Datenkapselung ist zu unterlassen. Mehrfachvererbung sollte vermieden werden, mit der Ausnahme von Schnittstellenklassen als Basisklassen. Schnittstellenklassen sind Klassen, welche ausschließlich *pure virtual* Methoden enthalten. Besondere Vorsicht ist bei Compiler-generierten Default-Konstruktoren und Zuweisungsoperatoren walten zu lassen.

Dynamischer Speicher

Die Verwendung von dynamischem Speicher (*new*, *delete*) ist in sicherheitskritischen Systemen bis auf wenige Ausnahmen verboten.

Ein Vergleich von MISRA-C++ und JSF C++

Aus der Sicht eines C++ Entwicklers ist ein Vergleich von MISRA-C++ und JSF C++ eine lohnende Übung, zeigt er doch interessante Ähnlichkeiten, Unterschiede und sogar Konflikte zwischen den beiden Codierungsstandards. Nebenbei schärft ein solcher Vergleich auch das Auge für mögliche Probleme die bei der Verwendung von C++ in sicherheitskritischen Systemen auftreten könnten.

MISRA-C++ und JSF C++ haben gemeinsame Wurzeln in MISRA-C. Beide übernehmen viele Regeln aus diesem Standard. Eine weitere gemeinsame Eigenschaft ist die restriktive Einstellung gegenüber Makros. Defensive Programmiertechniken werden von beiden Standards gefordert; der Einsatz von Tools zur statischen Source Code Analyse empfohlen.

MISRA C++ enthält keine Regeln bezüglich Codestil oder Codemetriken (im Vorwort wird jedoch die Verwendung von Stilrichtlinien vorgeschlagen). Im Gegensatz dazu hat JSF C++ spezifische Regeln für Codestil und Metriken (z. B. Regeln 1, 3, 41 – 45, etc.).

In einigen Punkten widersprechen sich MISRA-C++ und JSF C++. Beispielsweise verbietet JSF C++ den Einsatz von C++ Exceptions, nicht aber MISRA-C++. Beim Umgang mit Null-Pointern erfordert MISRA-C++ die Verwendung des *NULL* Makros, während JSF C++ die Verwendung des 0 Literals anstelle des *NULL* Makros empfiehlt.

Empfehlungen

C++ stellt viele Mechanismen bereit welche es einfacher machen robusten Code zu schreiben. Obwohl diese Mechanismen von den Codierungsstandards nicht explizit gefordert werden, sollten diese doch verwendet werden.

Beispielsweise hilft das RAII (Resource Acquisition Is Initialization) Idiom beim korrekten Verwalten von Ressourcen. RAII ermöglicht die Implementierung von Smart Pointern, welche viele Pointerfehler vermeiden können. Aufgrund der sehr eingeschränkten Verwendung von dynamischem Speicher in sicherheitskritischen Systemen sind Smart Pointer hier natürlich nur begrenzt nützlich. RAII ermöglicht aber auch die Implementierung von “scoped locks”, welche für die korrekte Implementierung von kritischen Abschnitten enorm hilfreich sind, da sie automatisch für das setzen und freigeben eines Mutex sorgen.

Ein wichtiges Feature für robusten C++ Code sind Templates. Sie ermöglichen z. B. die effiziente Implementierung von typsicheren Arrays mit Indexprüfungen zur Laufzeit. Eine weitere gute Anwendung von Templates ist die Implementierung von Festkommaarithmetik.

Beim Einsatz von Exceptions sollte ein C++ Entwickler mit dem Konzept der “exception safety” [5] vertraut sein. Exceptions können die Fehlerbehandlung in Programmen stark vereinfachen. Eine Exception zum falschen Zeitpunkt kann aber auch zu einem ungültigen Zustand von Objekten oder Strukturen führen, was Probleme zu einem späteren Zeitpunkt bringt. RAII ist hier das Mittel der Wahl.

Zusammenfassung

Der Einsatz von C++ bei sicherheitskritischen Systemen ist möglich, vorausgesetzt ein entsprechender Codierungsstandard wie MISRA-C++ oder JSF C++ wird verwendet und entsprechende Tools zur automatischen Überprüfung und Analyse des Source Codes werden eingesetzt. Ein Vergleich von MISRA-C++ und JSF C++ zeigt interessante Ähnlichkeiten, aber auch Unterschiede und Konflikte. Jeder Entwickler welcher den Einsatz von C++ in sicherheitskritischen Systemen plant sollte sich intensiv mit beiden Standards befassen.

Referenzen

[1] ISO/IEC 14882 International Standard: Programming Languages – C++, Second Edition, ISO/IEC, 2003

[2] MISRA-C: 2004 – Guidelines for the use of the C language in critical systems, MIRA Limited, 2004.

[3] Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, Lockheed Martin Corporation, 2005

[4] MISRA-C: 2008 – Guidelines for the use of the C++ language in critical systems, MIRA Limited, 2008.

[5] David Abrahams, Exception-Safety in Generic Components, http://www.boost.org/more/generic_exception_safety.html

Zum Autor

Günter Obiltschnig ist Gründer und Geschäftsführer der Applied Informatics Software Engineering GmbH, einem Software-Unternehmen spezialisiert auf Tools und Dienstleistungen rund um C++. Er verfügt über mehr als 16 Jahre Erfahrung in der Entwicklung von Software für verschiedenste Systeme – von verteilten Unternehmensapplikationen bis zu Embedded Systemen. Als Referent ist er regelmäßig auf verschiedenen internationalen Konferenzen wie z. B. der Embedded World vertreten.