Speicherhunger und Leistung von C++ Applikationen

Andreas Pfeiffer

C++ hat in den letzten Jahren immer mehr an Popularität gewonnen und wird sich wohl als die Standard-Programmiersprache der näheren Zukunft etablieren. Die Programmiersprache C wird überall bei der Programmierung von Embedded und Echtzeit-Applikationen verwendet. Es überrascht daher kaum, daß sich die Entwickler dieser Systeme auch vermehrt für den Einsatz von C++ interessieren.

Parallel zum steigenden Interesse der Anwender war die Sprache C++ einer bedeutenden Evolution unterworfen. Neue Features wie Templates (parametriesierbare Typen), Typ-Identifikation zur Laufzeit und Exception-Handling wurden in den Sprachumfang integriert. Diese Erweiterung geschah unter den Bemühungen einer Standardisierung durch das ANSI-Komitee. Ein stabiler Standard wird für 1996 oder 1997 erwartet.

Führende Compiler-Hersteller bemühten sich mit großem Entwicklungsaufwand die C++ Compiler-Technologie zu erweitern und Konformität zur Sprachdefinition im Annotated Reference Manual (ARM) zu erreichen. Microtec Research, Spezialist im Bereich der Cross-Compiler baute seine Compiler zuerst auf Basis von Cfront (einem C++ auf C-Übersetzer von AT&T) auf. Auf Grund der jüngsten Entwicklungen wurde jedoch die Technologie geändert und ein neuer Ansatz für den C++ Compiler der nächsten generation gemacht.

Wie effizient ist C++ im Vergleich zu C?

Über die Vorteile von C++ und der objektorientierten Programmierung wurde mittlerweile schon sehr viel publiziert. Sehr wenig Informationen existieren jedoch, wenn es um die Effizienz von C++ geht. Anders als bei großen Host-Systemen, wo beinahe unbeschränkt Speicher zur Verfügung steht, muß bei der Entwicklung von Embedded Systemen aus Kostengründen mit eingeschränkten Resourcen gearbeitet werden. Viele Entwickler von Embedded Applikationen sind daher über den Speicherhunger und eventuelle Leistungseinbußen beim Einsatz von C++ besorgt.

C++ ist eine Übermenge von C und kann daher auch nur mit diesem Sprachumfang zur Entwicklung benützt werden. Unter diesen Umständen soll natürlich eine reine C-Anwendung kompiliert mit einem C++ Compiler zu keinerlei Overhead führen. C++ kann also als typensicheres C verwendet werden. Die meisten Entwickler interessieren jedoch gerade die Vorteile der objektorientierten Eigenschaften der Sprache wie Wiederverwendbarkeit und Wartbarkeit. Dies führt zum Einsatz neuer Sprachelemente, die zu erhöhtem Speicherbedarf führen können, was jedoch sehr stark von der Implementierung im Compiler-System abhängt.

Objektorientierte Programmiermethoden können auch zu Leistungseinbußen führen, wenn die Implementierung der Objekte beim Einsatz nicht genau verstanden wird. So ist es zum Beispiel wichtig zu wissen, wieviel Zeit ein Constructor oder Destructor für ein Objekt benötigt. In zeitkritischen Programmabschnitten muß der Programmierer lange Laufzeiten für das Erzeugen von Objekten vermeiden. Gute Programmiertechnik ist noch immer der beste Weg Leistungseinbrüche durch schlecht entwickelte Constructors und Destructors zu vermeiden.

Einfluß von C++ auf den Speicherbedarf

Einige der Spracheigenschaften von C++ wie Templates, Inline-Funktionen, Vererbungsmechanismen oder Virtuelle Funktionen können den Speicherbedarf für eine Applikation erheblich beeinflussen. All diese Eigenschaften werden sowohl in den Applikationsprogrammen als auch in C++ Klassen-Bibliotheken verwendet.

Templates

Templates bieten eine hervorragende Möglichkeit einen Quell-Code-Teil für viele Programmbereiche zu verwenden. Nur eine einzige Implementation einer Klasse oder Funktion kann für unterschiedliche Datentypen verwendet werden. Die Definition der Sprache C++ sagt jedoch nichts darüber aus, ob der Objekt-Code für einzelne Daten-Typen einmal oder mehrmals generiert werden soll. Dies wird dem Compiler-System überlassen und ist daher extrem abhängig von dessen Implementierung. Ein schlechter Compiler kann so zu erheblichem Speicherbedarf führen.

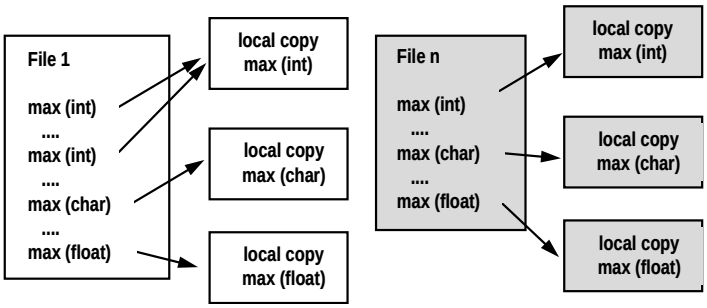

Bild 1: **Ineffiziente Implementierung von Templates**

Manche Compiler erzeugen im einfachsten Fall für jede Quelldatei und jeden dort verwendeten Datentyp eine Instanz der Template-Funktion (bzw. Klasse). In ***Bild 1*** ist dies dargestellt. Jede Quelldatei führt zu eigenen Instanzen der Funktion max. Die Konsequenzen dieses Ansatzes liegen auf der Hand. In einer Appliaktion mit 500 Quelldateien werden 1500 Kopien der Template-Funktion max generiert. Wenn der Linker diese Kopien einfach zusammen in die Objektdatei überführt, so ist diese erheblich größer als sie sein müßte. Es werden nämlich nur drei Kopien von max benötigt: Eine für int, eine für char und eine für float, wie in ***Bild 2*** gezeigt wird.

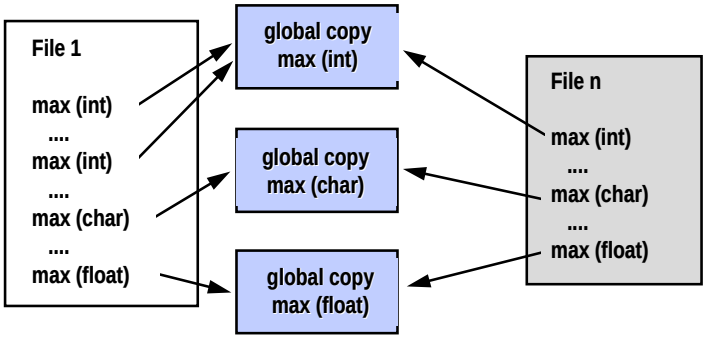

Bild 2: **Richtige Implementierung von Template-Funktionen**

In-Line Funktionen

Die Verwendung von In-Line-Funktionen kann auch zu unerwünschten Code-bereichen führen, die mehrfach redundant vorhanden sind. Das Problem hier ist jedoch weniger augenfällig als bei den Templates. Wenn eine Funktion mit dem Schlüsselwort inline definiert wurde, so wird bei jedem Aufruf dieser Funktion aus der Applikation eine Kopie dieser Funktion in den Objektcode kopiert. Eine Funktion, die innerhalb einer Klassendefinition definiert wurde, ist per Vorgabe eine Inline-Funktion. Der Ansatz ist, daß der Overhead bei Funktionsaufrufen weggelassen wird und dadurch die Performance steigt. Der Bedarf an Programmspeicher steigt jedoch mit der Größe der Funktion. Wenn der eigentliche Funktions-Code kleiner ist, als jener Bereich für den Einsprung und die Rückkehr benötigt wird, so sinkt der Speicherbedarf. Umgekehrt wächst er.

Leider gibt es dabei einen Haken: es ist dem Compiler nicht immer möglich eine In-Line-Funktion auch In-Line zu verwenden. Manchmal beruht das einfach auf schlechtem Programmierstil. Zum Beispiel könnte so eine Funktion rekursiv verwendet werden, oder jemand möchte die Adresse (`ptr=&foo()`) der Funktion feststellen. In diesen Fällen wird der Compiler die Funktion regulär verwenden. Ein größeres Problem stellt sich jedoch, wenn der Compiler selbst eine Funktion nicht als In-Line Funktion implementieren kann. Dies kann typischerweise durch die komplexe Flußkontrolle des Optimierers verursacht werden. In diesen Fällen wird die Funktion ebenfalls regulär implementiert.

Wenn nun eine In-Line Funktion nicht In-Line verwendet werden kann, ist diese per Vorgabe als statisch definiert. Für jede Datei, die auf diese Funktion verweist wird eine eigene Kopie der Funktion generiert. Jetzt hat man plötzlich den doppelten Nachteil. Die erwünschte Leistungssteigerung durch den Einsatz als In-Line Funktion fällt weg und die Code-Größe wird unnötigerweise wachsen, weil eine Kopie dieser Funktion in für jede aufrufende Quelldatei generiert wird.

Virtuelle Funktionen

Für unsere Effizienzbetrachtungen muß natürlich auch der Blick auf den Datenbereich einer Applikation gelenkt werden. Template-Objekte können zu mehrfach vorhanden Datenbereichen führen. Aber auch die Verwendung von Virtuellen Funktionen, besser gesagt die Virtual Function Tables können dazu führen, daß Datenbereiche redundant generiert werden.

Der Aufruf von virtuellen Funktionen geschieht über Tabellen mit Zeigern, den Virtual Function Tables. Diese Tabellen erzeugt der Compiler, damit zur Laufzeit die richtigen Funktionen ausgewählt werden können. So ist es natürlich nicht sehr effizient, wenn für jede Quelldatei in der virtuelle Funktionen aufgerufen werden eigene Function Tables generiert werden. Der bessere Weg ist es, wenn der Compiler nur dort die Tabelle erzeugt, wo die Funktion definiert wurde. Dies könnte natürlich immer noch zu redundanten Informationen führen, wenn die virtuelle Funktion eine Inline- oder Template-Funktion ist. Diese Vervielfältigung von Virtual Function Tables erhöht die Gefahr des übermässigen Speicherbedarfs.

Vererbung

Das Layout von abgeleiteten Objekten beeinflußt natürlich auch den Bedarf an Speicherplatz. Hier können Einsparungen erzielt werden, wenn das Objektlayout und die Tabellen für virtuelle Funktionen optimiert werden. C++ Implementierungen die auf Cfront basieren können bei manchen abgeleiteten Objekten zu sehr schlechten Ergebnissen führen. In ***Bild 3*** ist als Beispiel ein Objekt MDerived dargestellt, daß von mehreren Klassen abgeleitet wurde, die ihrerseits von einer virtuellen Basisklasse abstammen.

```
Class BaseClass { ... } ;
Class DerivedA : virtual public BaseClass { ... } ;
Class DerivedB : virtual public BaseClass { ... } ;
Class DerivedC : virtual public BaseClass { ... } ;
Class DerivedD : virtual public BaseClass { ... } ;

Class MDerived : DerivedA, DerivedB, DerivedC, DerivedD { ... } ;
```

Bild 3: **Beispielcode Mehrfache Vererbung**

In Cfront-basierenden Lösungen können Klassen, die von n Klassen mit der gleichen virtuellen Basisklasse im besten Fall zu n-1 (manchmal auch n) Kopien der Mitglieder der virtuellen Basisklasse führen (***Bild 4***).

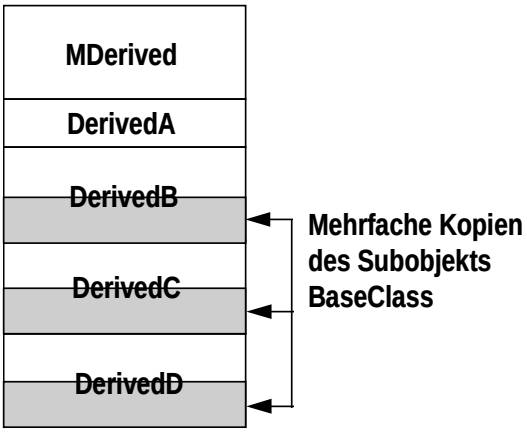

Bild 4: **Ineffiziente Implementierung von mehrfacher Vererbung**

Die effiziente Implementierung benötigt die Mitglieder der virtuellen Basisklasse nur einmal. Da Instanzen einer solchen Klasse zur Laufzeit erzeugt werden, betrifft dieser Bereich die Ausnutzung von wertvollem RAM-Speicher.

Produktivitätseinbußen durch überflüssige Debug-Informationen

Die Minimierung des notwendigen Platzbedarfs für eine Applikation ist wichtig, um die Kosten des Endprodukts in den vorgegebenen Grenzen zu halten. Das Problem der mehrfach vorhandenen Debug-Informationen hat jedoch im Besonderen auch Einfluß auf die Produktivität des Entwicklungsprozesses. Dieses Problem war auch schon bei den C-Entwicklungswerkzeugen bekannt. Beim Einsatz von C++ hat dieser Effekt jedoch sehr an Bedeutung gewonnen, da der Programmierstil hier dahin tendiert, in jeder Quelldatei sehr viele (oder alle) Header-Dateien einzubinden. Die komplexen Verknüpfungen von Klassen machen es schier unmöglich von vornherein festzustellen, welche Header-Datei zur Übersetzung einer Quelldatei wirklich benötigt wird.

Die alten C++ Compiler haben für jeden Typ der in irgendeiner Header-Datei deklariert oder definiert wurde Debug-Informationen generiert. Dies war unabhängig davon, ob der Typ dann wirklich in der Quelldatei verwendet wurde. Es liegt auf der Hand, daß dieser Weg, gerade wenn immer alle Header-Dateien mit eingebunden werden, zu einem gewaltigen Anwachsen der Debug-Informationen führen kann. Die Produktivität kann dadurch in mehreren Bereichen beeinflußt werden: Die Linker-Laufzeiten werden erheblich länger sein, da immer alle Debug-Informationen durch den Linker berücksichtigt werden müssen. Die Ladezeiten für den Debugger werden ebenfalls länger dauern. Der Debugger wird dann die überflüssigen Informationen ohnehin ignorieren. Schließlich kann natürlich auf dem Plattenspeicher gar nicht mehr genug Platz vorhanden sein, um sämtliche Daten darauf abzulegen.

Neue Wege für einen effizienten Einsatz von C++

Auf Basis des neuen C++ Toolsets von Microtec Research soll gezeigt werden, wie die oben genannten Punkte in ein effizientes Compilerdesign einfließen können. Die wesentlichen Eigenschaften sind:

- Schnellere Compile-, Link- und Ladezeiten
- Keine mehrfach vorhandenen Debug-Informationen
- Keine vervielfältigten Subobjekte aus Basisklassen
- Keine mehrfachen Template- und In-Line-Funktionen
- Vermeidung redundanter Virtual Function Tables

Schnellere Compile- und Linkzeiten

Um die Durchlaufzeiten im Compiler zu verbessern, wurde die Übersetzung von C++ in C beseitigt. Direkte Übersetzung führt zu Zeiteinsparungen im Bereich von 30%. Der Compiler unterscheidet ganz klar zwischen notwendigen und überflüssigen Debug-Informationen. Es werden nur für jene Variablen oder Funktionen Informationen erzeugt, die in der Quelldatei verwendet werden. Dies führt zur Verkleinerung der Ergebnisdatei in der Größenordnung von 50%. In einem sehr extremen Testfall erreichte man hier sogar Einsparungen von 92%!

Die größten Vorteile dieser Optimierung treten im Anschluß an den Compiler auf. Die Laufzeit des Linkers ist erheblich kürzer, weil nur mehr kleine Dateien verarbeitet werden müssen. Da der Linker sehr oft auf Dateien zugreifen muß, führt jede Reduzierung von Dateigrößen zu schnelleren Durchläufen. Für den oben erwähnten Fall mit 92% Platzeinsparung lief der Linker nur für 1/5 der ursprünglichen Laufzeit.

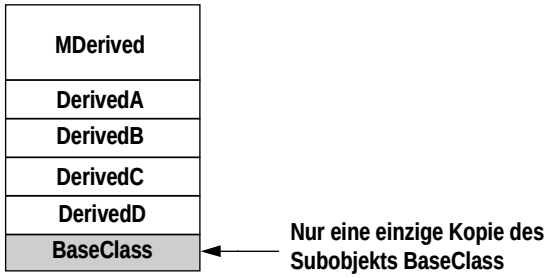

Bild 5: **Neues Objektlayout beinhaltet nur eine Kopie der Basisklasse**

Obwohl der Compiler überflüssige Debug-Informationen vermeidet, kann es trotzdem vorkommen, daß Funktionen oder Variablen in mehreren Modulen verwendet werden. Der Compiler wird für jedes dieser Module weiterhin eigene Debug-Informationen erzeugen. Der Linker sorgt jedoch dafür, daß auch hier redundante, d.h. mehrfach vorhande-

ne Debug-Informationen beseitigt werden Man spricht von einem optimierenden Linker.

Die Größe der Ergebnisdatei wird dadurch weiter reduziert und beschleunigt dadurch den Ladeprozeß in den Debugger. Für eine typische Applikation führten die Optimierungen in Compiler und Linker zur Verkürzung der Zyklus-Zeit für Editieren, Compilieren und Debuggen von 40%.

Weniger Platzbedarf

Mehrfach vorhandene Code- und Datenbereiche müssen sowohl durch den Compiler als auch durch den Linker vermieden werden. Der Linker ist das richtige Werkzeug um über Dateigrenzen hinaus zu optimieren.

Neue Optimierungsmethoden generieren Tabellen für virtuelle Funktionen nur in jener Datei, wo diese Funktion definiert wurde. Das Layout von Virtual Function Tables wurde gegenüber der Cfront-Implementierung ebenfalls verbessert.

Das neue Layout vermeidet das oben erwähnte Problem jener Objekte, in denen mehrfach die gleichen Funktionen virtueller Basisklassen vorkommen. Bild 5 zeigt diese Verbesserung. Dies kann zu erheblichen Einsparungen beim RAM-Verbrauch führen, wenn sehr viele Objekte über mehrfache Vererbung und virtuellen Basisklassen erzeugt wurden.

Der Linker optimiert vielfache Kopien von Template- und In-Line-Funktionen. Außerdem werden mehrfach vorhandene Tabellen für virtuelle Funktionen, erzeugt durch In-Line-, Template- oder virtuelle Funktionen, beseitigt. Viele Optimierungen, die früher im Compiler vorgenommen wurden und zu erheblichen Laufzeiten geführt haben, wurden in den Linker verlegt. Als Beispiel: Cfront benötigte sechs Stufen, nur um redundante Template-Funktionen zu vermeiden:

1- Übersetzen von C++ nach C (ausgenommen Templates)
2- Kompilieren von C
3- Prelinker stellt fest wo Templates verwendet werden
4- Übersetzen der Templates von C++ nach C
5- Kompilieren der Templates
6- Linken der Applikation

Cfront erzeugt beim Übersetzten der Quelldatei noch keine Instanzen für die Template-Funktionen. Statt dessen wird ein spezieller Prelinker eingesetzt der feststellt, welche Dateien ein bestimmtes Template verwenden. Cfront ruft anschließend den Compiler auf, um eine einzige Kopie zu erzeugen, die dann von allen Dateien verwendet wird. Wenn alle Template-Funktionen erzeugt wurden, erfolgt der endgültige Linker-Durchlauf.

Die Optimierungsalgorithmen im neuen Linker sorgen dafür, daß redundante Code- und Datenbereiche beseitigt und zusammengelegt werden. Diese Optimierung betrifft auch die Beseitigung überflüssiger Debug-Informationen. Der Compiler kennzeichnet die gewollten Mehrfach-Kopien, wie z.B. statische Funktionen, die der Linker dann im Anschluß nicht wegoptimiert.

Der Optimierungsansatz im Linker verlängert die Durchlaufzeiten kaum. Ein Linker greift intensiv auf Dateien zu und in diesem Fall werden überflüssige Code- und Datenbereiche beseitigt und nicht neuerlich in Dateien abgelegt. Ein Compiler-basierender Ansatz würde erheblichen Aufwand bedeuten, die Abhängigkeiten zwischen den Dateien zu erkennen und auf dieser Basis zu optimieren. Die Laufzeiten wären erheblich länger.

Zusammenfassung

Der steigende Einsatz von C++, speziell auch in Embedded Applikationen, erfordert neue Compilertechnologien um kompakte Code- und Datengrößen zu erreichen, den Speicherbedarf zu verringern und den Entwicklungszyklus zu beschleunigen. Die Optimierungsmethoden herkömmlicher C-Compiler versagen beim Einsatz von C++. Die Optimierung muß speziell an die Anforderungen der neuen objektorientierten Eigenschaften angepaßt sein und den unnötigen Overhead, der sehr oft mit der Sprache C++ in Zusammenhang gebracht wird, verhindern. Nachdem viele C++ Applikationen auch besonders umfangreich sind, kann durch eine neue Compiler-Technologie die Zykluszeit beim Entwickeln beschleunigt werden. Die neue C++ Technologie von Microtec Research reduziert die erforderlichen Zeiten beim Linken und Laden durch vermeiden von redundanten Debug-Informationen und beim Compilieren durch Weglassen des Zwischenschritts der Übersetzung von C++ Programmen nach C. □

Der HTML-Ratgeber

Heinrich.E.G.Bonin, Hanser-Verlag
1996, Preis: ATS 277,-

Dieter Reiermann

Liebenswürdigerweise hat mich der Hanser-Verlag eingeladen, den HTML-Ratgeber zu prüfen. Zur Gestaltung meiner Homepage kam mir dieses Buch gerade recht. Mein erster Eindruck - als ein der Materie noch weitestgehend Unkundiger - „Ich werde mir mehr Zeit nehmen müssen". Abends vor dem Schlafengehen wollte ich im HTML-Ratgeber nicht schmökern. Also kein Lesebuch.

Bei meiner späteren Arbeit habe ich das Buch allerdings noch sehr gut brauchen können. Die HTML-Befehle sind kapitelweise funktionell gegliedert und umfassend beschrieben. Es werden einige interessante Beispiele gezeigt.

Zum Inhalt

Einleitung

Beschreibt hauptsächlich die Intentionen des Autors (Datum: Winter 95/96). Das Buch behandelt HTML 2.0, wieweit HTML 3.2 berücksichtigt wurde, konnte ich nicht herausfinden.

Software

Nur über URL, Diskette wird auf Wunsch zugesendet.

1. Konstrukte

WWW (Hallo World, Syntax der Konstrukte, HTTP), Semantik der Konstrukte, Uniform Resource Locator

2. Konstruktionen

Dynamisches Dokument, Dialog mit Formular, Präsentationen mit Viewer Applikationen, WWW-Server, Realisierung von Sicherheit

3. Konstruktionsempfehlungen

Organisation des Datenmaterials, Gestaltung des einzelnen Dokumentes, Systematisches Testen, Wartung und Betrieb

Anhang

Quellen, Abkürzungen, Index, Abbildungsverzeichnis, Tabellenverzeichnis

Autor

Prof.H.Bonin, Fachhochschule Nordostniedersachsen, Fachbereich Wirtschaft, Lüneburg, `http://cl3.fbw.fh-lueneburg.de:6667/`

Umschlaggestaltung

Das wohlbekannte Layout des Hanser-Verlag, aber was hat HTML mit Heidelbeeren zu tun?

Druck

etwa 10 Punkt, starke Gliederung durch Zeichensatz - vielleicht wirkt der Text dadurch etwas unruhig. Zahlreiche Bildschirmshots von HTML-Dokumenten.

Zusammenfassend

Nicht für „bloody beginners", als Arbeitsbehelf sehr gut geeignet. Aufbau: Ähnlich einer Sprachreferenz.