C++ Performance-Fallen und wie man sie umgeht

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

Einleitung

C++ [1] ermöglicht die Implementierung von sehr effizientem Programmcode. Ein gut implementiertes C++ Programm sollte niemals langsamer sein als ein äquivalentes C Programm – im besten Fall sogar schneller. Dennoch ist es sehr leicht möglich, uneffizienten C++ Code zu schreiben. Während uneffizienter C++ Code auf einem modernen PC vielleicht gar nicht auffällt, kann er auf einem leistungsmäßig eingeschränkten Embedded System fatal sein.

Der durch C++ mögliche hohe Abstraktionsgrad verschleiert oft die Sicht auf die Dinge, die "hinter den Kulissen" passieren. Speziell beim Einsatz der C++ Standard Bibliothek können unliebsame Überraschungen auftreten wenn man die interne Funktionsweise der Klassen und Algorithmen nicht berücksichtigt. Auch das Design von Klassen und Funktionen hat Auswirkungen auf die Performance.

Die Basics – Call By Value und Call By Reference

Im Gegensatz zu Sprachen wie z. B. Java, bei denen Objekte als Argumente bei einem Funktions- oder Methodenaufruf immer per Referenz übergeben werden, können bei C++ Objekte entweder als Wert („by value“) oder per Referenz („by reference“) übergeben werden. Der Unterschied kann, was die Laufzeit eines Programmes betrifft, gravierend sein. Folgendes Beispiel soll dies verdeutlichen:

```
void f1(std::vector<std::string> arg)
{
    // ...
}

void f2(const std::vector<std::string>& arg)
{
    // ...
}
```

Bei der Funktion *f1* wird der Vektor als Wert übergeben, bei *f2* als konstante Referenz. Beim Aufruf von *f1* muss der gesamte Vektor kopiert werden. Dies bedeutet im Detail:

- Für die Kopie des Vektors muss Speicher auf dem Heap allokiert werden.
- Jedes Element des Vektors muss einzeln in den neuen Vektor kopiert werden. Da es sich bei den Vektor-Elementen um Strings handelt, muss wiederum für den Inhalt jedes Strings Speicher auf dem Heap allokiert werden, und der Inhalt des Strings muss kopiert werden.

Im Gegensatz dazu wird beim Aufruf von *f2* einfach die Adresse des Vektors übergeben. Dies kann bei vielen Prozessor-Architekturen, bzw. Compilern effizient über ein Register erfolgen. Der Performance Unterschied zwischen einem Aufruf von *f1* und *f2* sollte einleuchtend sein; speziell wenn man sich vorstellt, dass der übergebene Vektor vielleicht 100000 Elemente enthält. Nun ist dieses Beispiel sicher extrem; trotzdem findet man solchen Code bisweilen vor. Vielleicht wird einfach das

& in der Deklaration des Arguments vergessen (was auch bei erfahrenen Programmierern vorkommen kann), oder ein z. B. aus der Java Welt kommender unerfahrener C++ Entwickler ist sich des Problems gar nicht bewusst.

Dynamischer Speicher – der größte C++ Performance Killer

Dies führt gleich zum größten Performance Killer in C++ – der dynamischen Speicherverwaltung. Allokationen und Deallokationen von dynamischem Speicher auf dem Heap sind in C++ relativ aufwändig. Dies ist bedingt durch die recht komplexe Verwaltung des Heaps. Ein Aufruf von *new* ist in C++ wesentlich aufwändiger als in Sprachen, bzw. Laufzeitsystemen mit automatischer Garbage Collection wie z. B. Java und .NET. Bei diesen Sprachen besteht die Allokation von Speicher am Heap im Wesentlichen daraus, den Zeiger auf das aktuelle Ende des benutzten Heaps um die benötigte Anzahl an Bytes zu verschieben, vorausgesetzt genügend freier Speicher ist vorhanden. Die schnelle Allokation geht allerdings zu Lasten der Garbage Collection, die relativ viel Zeit benötigen kann, wenn sie erst einmal in Aktion treten muss. Bei C++ (wie auch C) muss hingegen eine Freispeicherliste verwaltet werden. Bei mehreren Threads muss der Zugriff auf den Heap zusätzlich synchronisiert werden. Zusätzlich ergibt sich speziell im Embedded Bereich (begrenzter Speicher und lang laufende Programme) das Problem der Heap Fragmentierung. Heap Fragmentierung tritt bevorzugt auf wenn abwechselnd große und sehr kleine Objekte mit unterschiedlicher Lebensdauer allokiert und deallokiert werden. Irgendwann wird der Heap durch die vielen kleinen Objekte so unterteilt, das keine großen Objekte mehr allokiert werden können, obwohl eigentlich noch genug freier Speicher verfügbar wäre (siehe Abbildung 1). Ein an Java oder .NET angelehnter Programmierstil, wo ständig Objekte am Heap erzeugt und wieder zerstört werden ist für die Performance eines C++ Programms katastrophal. Es empfiehlt sich daher, wann immer möglich, C++ Objekte am Stack zu erzeugen. Dies erfordert lediglich das Umsetzen des Stack Pointers und erzeugt somit keinen Aufwand für die Freispeicherverwaltung. Aufgrund der Arbeitsweise von CPU Caches kann auch der Zugriff auf Stack Objekte schneller sein als der Zugriff auf Heap Objekte. Natürlich gibt es aber Situationen, wo Objekte am Heap erzeugt werden müssen – entweder wegen ihrer Lebensdauer, oder ihrer Größe. Hier kann sich die Verwendung vom Memory Pools sehr positiv auswirken. Dabei wird der Speicher für bestimmte Objekte, die ständig erzeugt und wieder zerstört werden, nicht an den Heap zurück gegeben, sondern in einer Liste verwaltet. Wird ein neues Objekt erzeugt, wird zunächst in der Liste nachgesehen, ob ein freier Speicherblock vorhanden ist, und dieser verwendet, bevor ein neuer Speicherblock auf dem Heap reserviert wird. Je nach Anwendungsbereich können die Speicherblöcke auch vorallokiert werden, oder die Anzahl der maximal allokierbaren Blöcke wird begrenzt. Eine Implementierung einer solchen Klasse findet sich z. B. in den POCO C++ Libraries [2]. Eine ähnliche Technik kann man auch anwenden, wenn in einer Echtzeit- oder sicherheitskritischen Applikation der Heap nicht benutzt werden darf.

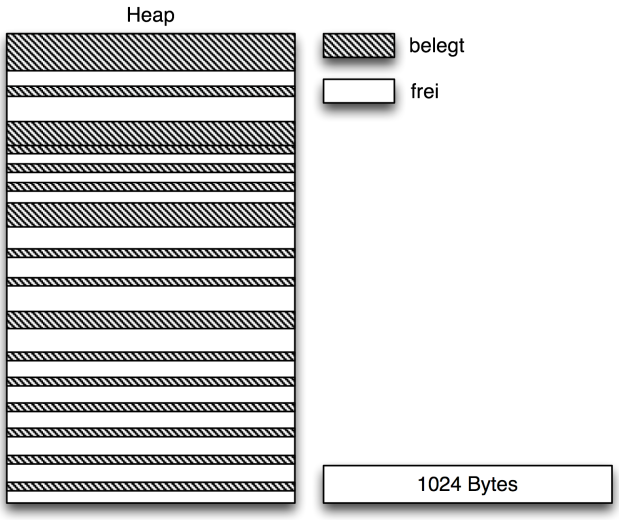

Abbildung 1: Heap Fragmentierung. Obwohl insgesamt noch genug freier Speicher vorhanden ist, kann kein zusammenhängender Block von 1024 Bytes mehr allokiert werden.

Strings und Vektoren

Die C++ Standardbibliothek bietet mit *std::string* eine leistungsfähige String-Klasse an. Diese kann jedoch schnell zu einer Performance-Falle werden, wenn man nicht aufpasst. Der Speicher für den Inhalt des Strings wird auf dem Heap angelegt (abgesehen von möglichen Optimierungen, wo kurze Strings bis zu 8 oder 16 Zeichen direkt im Objekt gespeichert werden). Bei der Implementierung von *std::string* kommen im wesentlichen zwei Techniken zum Einsatz. Bei der ersteren besteht die Möglichkeit, dass sich mehrere String-Objekte, die durch Kopieren oder Zuweisungen initialisiert wurden, den selben Heap Block für den Inhalt teilen. Dies funktioniert allerdings nur, solange der String nicht verändert wird. Sobald auch nur die Möglichkeit besteht, dass der String verändert werden könnte, muss eine private Kopie des Inhalts erzeugt werden. Von dieser Art der Implementierung erhoffte man sich eine bessere Performance, durch die Vermeidung unnötiger Kopien. Allerdings hat sich mittlerweile herausgestellt dass die erhofften Performance-Verbesserungen in der Praxis selten auftreten, da einerseits der Zugriff auf den gemeinsamen Inhalt thread-safe erfolgen muss und andererseits das Erzeugen von Kopien häufiger nötig ist als angenommen. Mittlerweile wird *std::string* eher wieder konservativ implementiert – bei jedem Kopieren des Strings wird auch der Inhalt sofort kopiert.

Arbeitet man mit *std::string*, so gilt es vor allem die Anzahl der Heap Allokationen zu reduzieren. Der Inhalt des Strings wird in einem Block fester Größe gespeichert. Wächst der Inhalt des Strings über diese Größe hinaus, muss ein neuer Speicherblock ausreichender Größe allokiert werden, und der aktuelle Inhalt in den neuen Block kopiert werden. Daraufhin wird der alte Speicherblock freigegeben – insgesamt eine sehr aufwändige Aktion. Kennt man von vornherein die ungefähre Größe eines Strings, der z. B. durch Verkettung von Teilstrings erzeugt wird, kann man von Anfang an einen Speicherblock ausreichender Größe reservieren. Dies erfolgt durch die Methode *reserve()*. Beispielsweise wird in folgendem Beispiel bei einer modernen Implementierung von *std::string* nur einmal Speicher allokiert, nämlich beim Aufruf von *reserve()*. Würde man den Aufruf von *reserve()* weglassen, könnten (je nach Implementierung) zwei oder sogar drei Allokationen am Heap erfolgen.

```
std::string str;
str.reserve(128); // reserve space for 128 chars
str = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. ";
str += "Proin malesuada tristique tellus interdum aliquet.";
```

Eine weitere gute Praxis ist es, Aufrufe jener Methoden von *std::string* zu vermeiden, welche selbst neue *std::string* Objekte liefern – z. B. *substr()*.

Anstatt:

```
std::string s1 = "Lorem ipsum dolor sit amet";
std::string s2;
...
s2 = s1.substr(0, 5);
```

schreibt man besser:

```
s2.assign(s1, 0, 5);
```

Während man in der ersten Variante schlechtestenfalls zwei Allokationen und zwei Kopieroperationen verursacht, kommt man in der zweiten Variante schlechtestenfalls mit einer Allokation und einer Kopieroperation aus. Im besten Fall spart man sich sogar die Allokation wenn *s2* bereits einen Heap Block ausreichender Größe für den Inhalt besitzt.

Überlegenswert ist die mehrmalige Verwendung eines String Objekts. Wird z. B. in einer Methode ein temporäres String Objekt benötigt, so wird bei jedem Aufruf der Methode ein neuer Speicherblock auf dem Heap für diesen String angelegt. Wird dieser String hingegen als Member-Variable im Objekt gespeichert so kann der selbe String immer wieder benutzt werden und im besten Fall wird nur einmal für diesen String Speicher allokiert. Diese Optimierung funktioniert allerdings nicht, wenn die Methode von mehreren Threads gleichzeitig aufgerufen werden kann.

Für *std::vector* gilt sinngemäss das gleiche. Auch hier führt die überlegte Verwendung von *reserve()* zu signifikanten Performance-Verbesserungen.

Zusammenfassung

Obwohl C++ prinzipiell eine Programmiersprache ist, die hoch performanten Code ermöglicht, und aus diesem Grund ja auch eingesetzt wird, gibt es doch viele Gelegenheiten, in Performance-Fallen zu tappen. Einige dieser Fallen wurden in diesem Artikel vorgestellt, wobei festgestellt wurde, dass die grundlegende Ursache für viele Performance-Probleme in der unüberlegten Benutzung des Heaps, bzw. in unnötigen Kopieroperationen liegt.

Referenzen

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

[2] POCO C++ Libraries
http://pocoproject.org

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 18 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. dem ESE Kongress und der Embedded World vertreten.