Andreas Döring[1]

Wie strukturiert man eine Programmbibliothek? Ein modularer Ansatz

Konzeptaufsatz zum Doktorandenworkshop des Instituts für Informatik der FU Berlin (Oktober 2003)

Überblick

In der Bioinformatik spielt die Verarbeitung von Sequenzen eine große Rolle. Mit dem Projekt SEQAN wird das Ziel verfolgt, effiziente Algorithmen zur Sequenzanalyse in einer einheitlichen C++ Bibliothek zusammenzufassen. In der jetzigen Phase des Projekts ist es zunächst einmal vorrangig, sich über eine geeignete Struktur für den Aufbau der Bibliothek klar zu werden. Hier soll ein modularer Ansatz vorgestellt werden, der gegenüber alternativen Strategien deutliche Vorteile in Bezug auf Flexibilität und spätere Erweiterbarkeit der Bibliothek bietet.

Eine C++ Bibliothek zur Sequenzanalyse

Die Verarbeitung von Sequenzen spielt in der Bioinformatik eine wichtige Rolle. RNA und DNA, die Träger der Erbinformation in lebenden Zellen, sind lange Ketten aus immer den gleichen vier Molekülbausteinen. Sie werden durch komplizierte Zellprozesse abgelesen und in Proteine übersetzt („Translation“), die wiederum Ketten von Molekülen („Aminosäuren“) sind. Ein simples Modell zur Beschreibung derartiger Molekülketten sind einfache Strings: Die in einem DNA-Stück liegende Information kann beispielsweise als String über einem vier-elementigen Alphabet {A, C, G, T} aufgefasst werden. Ist die richtige Reihenfolge der Basenpaare einer DNA erst einmal bestimmt und liegt in Form eines Strings vor, kann dies als Grundlage für vielfältige Analysen dienen.

In den letzten Jahren ist die Zahl der entschlüsselten Gensequenzen sprunghaft angestiegen. Nachdem inzwischen die Genome ganzer Organismen, unter anderem auch das des Menschen, sequenziert worden sind, steht die Wissenschaft nun vor der Herausforderung, dieser enormen Datenfülle Herr zu werden. Es ist die Aufgabe der Bioinformatik, die dafür notwendigen Algorithmen bereitzustellen und für Forschung benötigten Softwaretools zu entwickeln. Ziel unseres Projektes mit dem Arbeitstitel SEQAN ist die Entwicklung einer Programmbibliothek effizienter Algorithmen und abstrakte Datenstrukturen in der Programmiersprache C++ für die Verarbeitung großer Mengen biomedizinischer Daten, insbesondere für die Analyse von Gen- und Proteinsequenzen. Dazu gehören Algorithmen, die eine schnelle Suche von String oder Stringmustern in langen Texten zulassen. Um die Suche zu beschleunigen werden in der Praxis oft spezielle Datenstrukturen (z.B. Suffixbäume) eingesetzt, die ebenfalls implementiert werden sollen. Eine weitere Gruppe von Algorithmen zur Stringanalyse betreffen die Suche nach ähnlichen Strukturen (so genannte „Motive“) in zwei oder mehreren Strings, sowie das Alignment von Strings. Außerdem soll SEQAN möglichst auch die Anbindung an bestehende biologische Datenbanken und verschiedene zur Speicherung von Sequenzen übliche Dateiformate unterstützen sowie detaillierte statistische Informationen über gegebene Sequenzen bereitstellen können.

Mit SEQAN soll zum einen die schnelle und effiziente Entwicklung von Prototypen für neue Softwaretools ermöglicht werden, und zum anderen böte sich die Bibliothek auch als Testfeld für aussagekräftige Vergleiche verschiedener Algorithmen und Programme unter realistischen

[1] Arbeitsgruppe „Algorithmische Bioinformatik“ der FU Berlin. doering@inf.fu-berlin.de

Bedingungen an. Außerdem soll SEQAN, sobald die Entwicklung weit genug vorangeschritten ist, gezielt in der Anwendung auf aktuelle Fragestellungen der Bioinformatik und Biomedizin zu erproben werden.

SEQAN soll folgende Kerneigenschaften aufweisen:

- **generisch**: In einem STL-artigen Programmierstil soll die Bibliothek ihre Funktionalität über möglichst universelle Schnittstellen bereitstellen, bei denen von der eigentlichen Implementierung weitgehend abstrahiert worden ist. Die Bibliothek ist dadurch einfacher zu handhaben und die Benutzung leichter zu erlernen, und es wird möglich, ohne große Änderung des Programmcodes je nach Situation verschiedene Implementierungen auszuprobieren.
- **effizient**: Mit SEQAN soll es möglich sein, schnell, einfach und übersichtlich zu programmieren, ohne dabei Opfer bezüglich Laufzeit und Platzbedarf des fertigen Programmes erbringen zu müssen
- **portabel**: SEQAN wird konform zu üblichen C++-Standards programmiert und auf unterschiedlichen Compilern und Betriebssystemen lauffähig sein. Dies wird bereits während der Entwicklung durch ständige Tests auf unterschiedlichen Plattformen sichergestellt.
- **erweiterbar und modular**: SEQAN soll modular aufgebaut werden: Verschiedene Module können vom Benutzer frei miteinander kombiniert und zu neuen Klassen gebündelt werden. Außerdem ist es möglich, neue Module nahtlos in das bestehende System zu integrieren. SEQAN verwendet dabei einen speziellen Weg für die Koppelung der Module untereinander, der im Folgenden diskutiert werden soll.

Das Problem: Spezialisierung von Funktionen

Bei der Konzeption einer Klassen- oder Template-Bibliothek wird man mit der Frage konfrontiert, auf welche Weise sich die Funktionalität, die in der Bibliothek implementiert werden soll, am besten auf unterschiedliche Klassen aufteilen lässt, und zwar möglichst ohne dabei die Möglichkeiten für eine spätere Erweiterung der Bibliothek oder eine Optimierung des Codes einzuschränken. Dass man es hier keinesfalls mit einem trivialen Problem zu tun hat, soll anhand des folgenden Beispiels aus der Sequenzanalyse verdeutlicht werden:

Strings sind Ketten von Zeichen aus einem festen Alphabet, und bestimmte Operationen auf Strings können direkt auf der Grundlage entsprechender Operationen auf dem Alphabet beschrieben werden. Ein Vergleich von zwei Strings beispielsweise könnte etwa wie folgt implementiert werden:

```
bool string_compare (str1, str2)
{
      if (str1.length() != str2.length())
      {
            return false;
      }

      for (i = 0; i < str1.length(); ++i)
      {
            if (! alphabet_compare(str1[i], str2[i]))
            {
                  return false;
            }
      }
      return true;
}
```

Dieses Programm funktioniert für alle Alphabete, zu denen es eine passende Vergleichsfunktion „alphabet_compare“ gibt. Es liegt nahe, Stringfunktionen wie „string_compare“ in einer Stringklasse S und Alphabetfunktionen wie „alphabet_compare“ für jedes Alphabet in einer Alphabetklasse A zu implementieren. S könnte A als Template-Argument erhalten, dann wäre S<A> eine Klasse, die Stringfunktionen zum Alphabet A exportiert.

Nun ist das gerade vorgestellte Programm zum Vergleich zweier Strings zwar sehr vielseitig anwendbar, es gibt jedoch in den meisten Fällen viel schnellere Alternativen. Besteht ein Alphabet nur aus wenigen Buchstaben, so könnte man mehrere Zeichen pro Maschinenwort abspeichern und dann alle auf einmal vergleichen. Auch könnten hier eventuell spezielle Prozessorinstruktionen für den Vergleich von Speicherblöcken zur Anwendung kommen, die unter Umständen viel schneller sind als eine Programmschleife. In vielen Fällen wäre es daher sicher wünschenswert, die Funktion „string_compare“ je nach Alphabet durch eine optimierte Version ersetzen zu können.

Eine Möglichkeit, dies zu erreichen ist die Verwendung von Template Spezialisierungen: Die Memberfunktion S<A>::string_compare der generische Templateklasse S<A> wird für ein spezielles Alphabet myA durch eine Funktion S<myA>::string_compare ersetzt. Dieser Ansatz hat allerdings den Nachteil, dass eine Spezialisierung für jede Alphabetklasse myA einzeln geschehen müsste. Sobald man S von einem weiteren Templateargument abhängig machen will, wäre man außerdem auf „Partielle Template Spezialisierung“ angewiesen, was vom C++ Standard zwar vorgesehen, jedoch noch nicht von allen gängigen Compilern unterstützt wird.

Ein anderer Weg, S<A>::string_compare zu ersetzen, bestände darin, von S<A> eine Klasse myS<A>:S<A> abzuleiten, und dann in myS<A> die alternative Funktion myS<A>::string_compare zu implementieren. Dieser Ansatz vermeidet zwar die Probleme der Template Spezialisierung, hat jedoch den folgenden Nachteil: Angenommen, in S<A> existierte z.B. eine weitere Funktion „string_search“, die string_compare verwendet, um den String zu durchsuchen. Nun würde S<A>::string_search zwar auf myS<A> vererbt werden, doch solange string_search nicht in myS<A> implementiert ist, kann string_search nicht myS<A>::string_compare aufrufen. Zwar gäbe es die Möglichkeit, string_compare als virtuelle Funktion zu deklarieren, doch dies wäre aus Gründen der Codeoptimierung wenig wünschenswert.[2]

Es gibt jedoch einen Trick, mit dessen Hilfe Funktionsaufrufe entgegen der Ableitungshierarchie stattfinden können, ohne dass dabei virtuelle Funktionen verwendet werden müssten. Dieser Trick soll nun vorgestellt werden.

Ein Cast zur Basisklasse

Das Problem besteht darin, dass S<A> als Basisklasse von myS<A> die abgeleitete Klasse nicht „kennt“, und daher aus S<A> heraus auch keine Funktionen von myS<A> aufgerufen werden können. Dies ändern wir einfach, indem wir S<A> als zusätzliches Template-Argument die abgeleitete Klasse mitgeben, also: myS<A>:S<A, myS<A> >. Der Aufruf von myS<A>::string_compare aus S<A, myS<A> >::string_search heraus findet nun wie folgt statt: In string_search wird der this-Pointer (der, da string_search eine Memberfunktion von

[2] Virtuelle Funktionen können vom Compiler nicht als inline Funktionen in den Code eingefügt werden. Daher kann auch nicht der Overhead vermieden, der bei jedem Funktionsaufruf unweigerlich entsteht. Zudem bedeutet die Verwendung von virtuellen Funktionen stets, dass das Objekt ein weiteres, verstecktes Datenmember mit dem Platzbedarf eines Pointers (gewöhnlich 4 Byte) bekommt.

S<A, myS<A> > ist, ein Pointer vom Typ S<A, myS<A> > * ist) mit einem statischen Cast in einen Pointer vom Typ myS<A>* verwandelt.[3] Über den so veränderten this-Pointer kann nun problemlos auf die Memberfunktionen von myS<A> zugegriffen werden.

Beispiel:

```
template <class TAlphabet, class TThis>
class S
{
      bool string_compare(...) {...}
      TAlphabet * string_search (...)
      {
            ...
            //Aufruf von string_compare
            static_cast< TThis *>(this)->string_compare(...)
      }
};

template <class TAlphabet>
class myS: public S<TAlphabet, myS<TAlphabet> >
{
      bool string_compare(...) {...}      //diese Funktion wird von
                                          //string_search aufgerufen
};
```

Die hier vorgestellte Template-Klasse S ist ein Beispiel für ein „Modul“, wie es in SEQAN verwendet werden soll: S erhält als eines seiner Template-Argumente den Typ „TThis“ einer Basisklasse von S, die wir als „Anker“ bezeichnen. Module alleine können nicht instanziiert werden, sie dienen lediglich als Basisklassen für Anker. In obigem Beispiel wäre myS ein Anker für S – doch es böte sich an, myS wiederum als Modul zu implementieren. In SEQAN wird alle Funktionalität durch Modulen gekapselt, wohingegen Anker lediglich dazu dienen, die gewünschten Module miteinander zu kombinieren.

Durch den Trick des Casts zur Basisklasse ergeben sich vielfältige Möglichkeiten der Interaktion zwischen den einzelnen Modulen. Dadurch erhält die Bibliothek eine hohe Flexibilität und bleibt gegenüber späteren Änderungen offen.

Literatur

[Zur Sequenzanalyse]

- Dan Gusfield, “*Algorithms on Strings, Trees and Sequences*“ (1997), Cambridge University Press
- Gonzalo Navarro, Mathieu Raffinot, “*Flexible Pattern Matching in Strings*” (2002), Cambridge University Press

[Zu C++]

- International Standard ISO/IEC 14882, “Programming Languages – C++“ (1998)
- Bjarne Stroustrup, “The C++ Programming Language” (first edition 1985, third edition 1991), Addison-Wesley
- Scott Meyers, “Effective C++” (second edition 1998), Addison-Wesley
- Scott Meyers, “More Effective C++” (1996), Addison-Wesley
- Scott Meyers, “Effective STL” (2001), Addison-Wesley
- Matthew H. Austern, “Generic Programming and the STL” (1998), Addison-Wesley

[3] Damit dieser Cast sicher ist, muss natürlich gewährleistet sein, dass S<A, myS<A> > tatsächlich als Basisclasse eines myS<A>-Objektes instanziiert worden ist.