Zur Beschleunigung von Compilerläufen durch Vereinfachung von Header-Includes

Einleitung

Viele Entwickler klagen über zu lange Compiler-Laufzeiten, die tagsüber durch lange Turnaround-Zeiten die Entwicklung behindern und nachts dazu führen, daß nicht der gesamte Code für alle Mitarbeiter übersetzt werden kann. Neben der Geschwindigkeit der Compiler und Maschinen oder der Anzahl und Größe der zu übersetzenden Dateien hängt die Compilierzeit aber auch davon ab, wieviel Code der Compiler bei einer einzelnen zu übersetzenden Datei vorgeführt bekommt. Dieses Dokument beschreibt Möglichkeiten, hier zu optimieren.

Das Problem

Von einer Quelldatei benutzte Typen, Funktionen und Macros werden typischerweise in Header-Dateien bereitgestellt, die von der Quelldatei includiert werden. Wie groß diese Dateien sind, oder welche anderen Header-Dateien dadurch „nachgezogen“ werden, ist dem Entwickler normalerweise nicht bekannt, oder er hat keinen Einfluß darauf. Hier muß er sich darauf verlassen, daß die angebotenen Header-Dateien „gut“ implementiert sind, was bedeutet

1. sie sind nicht größer als erforderlich, und
2. sie ziehen nicht mehr andere Header-Dateien als unbedingt erforderlich nach.[1]

Viele Header-Dateien werden aber auch von uns Entwicklern selbst geschrieben, und viele dieser Dateien sind derzeit nicht immer „gut“ im obigen Sinne – was im wesentlichen darauf zurückzuführen ist, daß für die Problematik kein Bewußtsein besteht. Dieses Bewußtsein wollen wir jetzt schaffen! Was für Punkte sollten also bedacht werden, um effizientere Header-Dateien zu schreiben? Wir wollen zwei Ziele betrachten:

+ so wenig andere Header-Dateien wie möglich nachzuziehen, und
+ Header-Dateien so klein wie möglich zu gestalten.

Vor allem der erste Punkt sorgt für eine spürbare Beschleunigung der Compiler.

So wenige andere Header-Dateien wie möglich nachziehen

Diese Beispiel ist teilweise entnommen aus dem Buch „Exceptional C++“ von Herb Sutter. Wenn ihr Zeit habt: lest es! Wenn ihr wenig Zeit habt: lest mindestens die Seiten 99 – 118!

```
/// \file Beispielheader
#include <iostream>   // Wegen std::ostream
#include "a.h"        // definiert Klasse A
#include "b.h"        // definiert Klasse B
#include "c.h"        // definiert Klasse C
class X
{
  public:
    A f( A a );
  private:
    B* b_;
    C c_;
};
std::ostream& operator<<( std::ostream& os, const X& x );
```

Was kann man hier nun optimieren?

---

[1] Es mag zwar bequem sein, wenn mit einer einzigen `#include`-Anweisung in einer `.cc`-Datei alle unter allen möglichen Umständen benötigten Klassen und Methoden deklariert werden, aber der Compiliergeschwindigkeit ist es eher abträglich.

Insbesondere bei Klassen ist es möglich, durch einfaches Deklarieren den Compiler zufriedenzustellen. Deklarieren ist einfaches Nennen eines existierenden Klassennamens (`class Contract;`) oder einer Funktion/Methode (`int getValue();`). Definieren ist das volle Sichtbarmachen einer kompletten Klassendefinition bzw. das Ausimplementieren einer Funktion/Methode. Zwischendinge zwischen Definieren und Deklarieren gibt es nicht.[2]

Die Beschränkung auf Deklarationen ist auch für einige Klassen der Standardbibliothek möglich. Für die I/O-Klassen existiert eigens eine Systemdatei `<iosfwd>`, die alle Klassen deklariert. Diese Datei ist **wesentlich** schneller zu parsen als die eigentlichen Definitionen der I/O-Klassen.[3]

Wir fangen einmal von der anderen Seite an und nennen die Situationen, bei denen auf ein `#include` *nicht* verzichtet werden kann.

Wann muß includiert werden?

1. Deklarationen von Systemfunktionen dürfen nicht in den Applikationscode geschrieben werden! Es ist immer die entsprechende System-Headerdatei einzubinden.
2. Wenn eine Klasse von einer anderen Klasse ableitet, muß die Definition der Basisklasse sichtbar sein.
3. Wenn eine Klasse einen Member vom Typ einer anderen Klasse enthält, muß die Definition der Klasse des Members sichtbar sein.
4. Wenn eine Methode/Funktion **definiert** wird, müssen die Typen der Parameter und der im Funktionskopf mit throw deklarierten Exceptions sichtbar sein. Dies gilt insbesondere auch für inline-Methoden/Funktionen! Wenn ein Parameter ein Pointer ist, muß der Typ, auf den gezeigt wird, nur dann definiert sein, wenn er innerhalb der Methode auch verwendet wird.
5. Wenn eine Methode die einen Pointer liefert überschrieben wird, und der dereferenzierte Rückgabetyp der überschreibenden Methode eine Klasse, die von dem Typ des dereferenzierten Rückgabetyps abgeleitet ist, liefert, dann müssen die dereferenzierten Rückgabetypen auch dann definiert sein, wenn die Methode nur deklariert wird.

In allen anderen Fällen reicht eine Deklaration der verwendeten Klasse/Funktion!

Das Beispiel kann also stark vereinfacht werden:

1. `#include <iostream>` ist **viel** zu schwer. Es würde bereits `ostream.h` genügen, und selbst das ist zu viel. Hier reicht das recht unbekannte `#include <iosfwd>`, das die I/O-Stream-Klassen vorwärtsdeklariert.
2. `#include "a.h"` kann ersetzt werden durch `class A`. Die Klasse A wird lediglich in einer Deklaration benutzt, dann braucht sie auch selbst nur deklariert zu werden.
3. `#include "b.h"` kann ersetzt werden durch `class B`. Es wird nur ein Pointer auf B verwendet; hierfür reicht immer eine Deklaration.

---

[2] Deklarationen an sich sind nicht schädlich, weil durch sie noch kein Code generiert wird oder sonstige Annahmen über die deklarierten Klassen gemacht werden. Das typsichere Linken unter C++ sowie der Vergleich Definition–Deklaration durch den Compiler garantieren schließlich die Konsistenz.

[3] std::vector kann nicht vorwärtsdeklariert werden, std::string schon, aber dafür gibt es keinen Systemheader. Vermutlich ist auch letzteres nicht sinnvoll, weil Strings fast immer auch wirklich *verwendet* werden.

4. `#include "c.h"` kann **nicht** ohne weiteres ersetzt werden durch `class C`, weil `C` direkt als Member benutzt wird.[4] Es ist aber in Einzelfällen zu überlegen, ob nicht der Compiliervorteil schwerer wiegt als der Laufzeitnachteil, wenn man statt der Einbettung einen Pointer benutzt. Kriterien sind:
   + wie oft wird `X` konstruiert/zerstört?
   + wie viele und wie große Header zieht `c.h` nach sich?

Unser Beispiel kann also (ohne die Optimierung in Punkt 4) so aussehen:

```
/// \file Beispielheader
#include <iosfwd>     // I/O-Streams vorwärtsdeklarieren
#include "c.h"        // definiert Klasse C
class A;
class B;
class X
{
  public:
    A f( A a );
  private:
    B* b_;
    C c_;
};
std::ostream& operator<<( std::ostream& os, const X& x );
```

Die Änderungen mögen nicht sehr spektakulär aussehen, aber die Beschleunigung des Compilers kann durchaus spektakulär sein – der optimierte Header kann ja in Hunderte von `*.cc`-Dateien einfließen. Natürlich müssen durch eine solche Änderung in den `*.cc`-Dateien, in denen die Methoden und Operatoren wirklich *implementiert* oder *benutzt* werden, die jeweils benötigten Dateien `a.h` und `b.h` dann direkt includiert werden. Dies dient aber auch der Klarheit des Codes.

Header-Dateien für im Code nicht mehr verwendete Funktionen müssen übrigens immer mit ausgebaut werden! Wird eine exotische Funktion aus dem Code ausgebaut, so sollte geprüft werden, ob eigens für diese Funktion ein Systemheader includiert wurde, und ob dieser jetzt obsolet ist. Es ist guter Stil, beim includieren von Headern zu vermerken, welche Klasse/Funktion bekannt gemacht werden soll.

Verkleinern von Header-Dateien

Wir gehen von der Situation aus, daß wir eine Bibliothek oder ein Framework schreiben oder warten wollen, das aus mehreren `*.cc`-Dateien besteht. Das Interface wird wie üblich per Header-Datei(en) den anwendenden Applikationen bekannt gemacht.

Die wesentliche Idee besteht darin, sich Gedanken über den Bereich zu machen, in dem eine Klasse/Funktion/Konstante verwendet wird.[5] Es gibt mindestens die Kategorien:

1. wird nur intern von der erstellten Bibliothek benutzt
2. wird oft vom externen Code benutzt
3. wird nur gelegentlich vom externen Code benutzt

Es ist naheliegend (aber in der Regel vom Entwickler nicht berücksichtigt), daß Definitionen der Kategorie 1 nicht in nach außen freigegebene Header gehören, und auch nicht durch solche Header includiert werden sollen! Dagegen müssen Definitionen der Klasse 3 zwangsläufig in freigegebene Header. Definitionen der Kategorie 2 sollten in eigene Header wandern, die vom externen Code nur bei Bedarf benutzt werden sollen.

Ein Beispiel:

---

[4] Genauer: Um die z.B. Größe der Klasse X zu bestimmen, muß der Compiler Informationen über die Größe der Klasse C besitzen.

[5] Die Ergebnisse dieser Gedanken können auch in Kommentaren festgehalten werden.

```
/// \file public.h
class RarelyUsed;         // Vorwärtsdekl. reicht
class AlwaysUsedImpl;     // Versteckte Implementierung
class  OftenUsed { ... }; // Oft benutzt, also hier definieren
class AlwaysUsed
{
  void  oftenCalledMethod(  OftenUsed & o );
  void rarelyCalledMethod( RarelyUsed & o );
  AlwaysUsedImpl* impl_;
};

/// \file rareClasses.h
class RarelyUsed {...};      // Hier definiert, weil selten gebraucht

/// \file private.h
class AlwaysUsedImpl {...};  // Hier definiert, weil Implementierungsdetail
```

Möglichkeiten zur Beschleunigung

Viele oben genannte Möglichkeiten zur Beschleunigung der Compiler sind einfach „da“. Andere muß man erst herbeiführen, was gelegentlich auf Kosten des Laufzeitverhaltens von DECIDE geht. Hier ist durch den einzelnen Entwickler sorgfältig abzuwägen.

- Durch reine Deklaration von Methoden (Verzicht auf inline-Inplementierungen) kann auf die Typen verzichtet werden, die in Interface und Implementierung der Methoden benutzt werden – zu lasten der Laufzeit.
- Member können als Pointer geführt werden; ggf. können sämtliche Member in einer Impl-Klasse untergebracht werden, auf die dann nur ein einziger Pointer gehalten wird. Dies erfordert aber leider Proxy-Methoden. [6]
- Vererbung kann in einigen Fällen durch Komposition ersetzt werden (oft ist dies auch eine bessere Modellierung), und dann wiederum durch führen eines Pointers als Member. So kann auch auf die Definition der ehemaligen Oberklasse verzichtet werden.

[6] fs-SmartPointer erfordern die Definition der referenzierten Klasse, sind also hierfür ungeeignet.