

Boost C++ Libraries

Johannes Diemke

Department of Computer Science
Learning and Cognitive Systems

Grundlagen

- Freie von Experten begutachtete, portable C++ Bibliothek
- Nützlich, unabhängig von speziellen Anwendungsgebieten
- Strebt Referenzimplementierung an
- Neuer C++0x Standard wird verschiedene Boost Bibliotheken enthalten
- Funktioniert auf fast allen modernen Betriebssystemen
- Bekannte Linux und Unix Distributionen wie Fedora, Debian und NetBSD umfassen vorübersetzte Boost Packages

Wieso sollte man Boost nutzen?

- Produktivität
- Beschleunigt anfängliche Entwicklung
- Weniger Bugs
- Verhindert es „das Rad neu zu erfinden“
- Langfristige Kosteneinsparungen bei der Wartung

Die Boost Header Organisation

- Alle Boost Header haben die Endung `.hpp`
- Sie befinden sich in dem Verzeichnis `boost/`
- Daher sehen alle Boost `#include` Direktiven folgendermaßen aus:

Listing 1: Boost Header

```
1 #include <boost/whatever.hpp>
2
3 int main() {
4     // use boost
5     return 0;
6 }
```

Die Boost Header Organisation (Forts.)

- Organisation ist nicht völlig einheitlich:
 - ▶ Einige ältere und die meisten sehr kleinen Bibliotheken haben ihre Header direkt in boost/
 - ▶ Die meisten Header der Bibliotheken sind jedoch in Unterverzeichnissen von boost/
Bsp.: `#include <boost/python/def.hpp>`
 - ▶ Einige Bibliotheken haben in boost/ aggregierte Header die alle Header einer Bibliothek einbinden
Bsp.: `#include <boost/python.hpp>`

Header-Only Bibliotheken

- Die meisten Boost Bibliotheken sind „Header-Only“
 - ▶ Bestehen komplett aus Header Dateien
 - ▶ Enthalten Templates und `inline` Funktionen
 - ▶ Es müssen keine separat kompilierten Bibliotheken gelinkt werden
- Es existieren einige Bibliotheken die separat übersetzt und gelinkt werden müssen
 - ▶ Sollten diese benötigt werden, so müssen diese in die Makefiles vom TORF Quellcode aufgenommen werden

Ein Minimalbeispiel

Listing 2: main.cpp

```
1  #include <iostream>
2  #include <boost/filesystem.hpp>
3
4  int main() {
5      boost::filesystem::path path("/usr/include");
6      bool result = boost::filesystem::is_directory(path);
7
8      std::cout << "Path is a directory : "
9                << std::boolalpha
10               << result << std::endl;
11
12     return 0;
13 }
```

Ein Minimalbeispiel (Forts.)

- Kompilieren und Linken führt zu unaufgelösten Abhängigkeiten
\$ `g++ main.cpp -o main`
- Benötigt die Bibliothek `libboost_filesystem.a`

Einschub: Bibliotheken (static libraries)

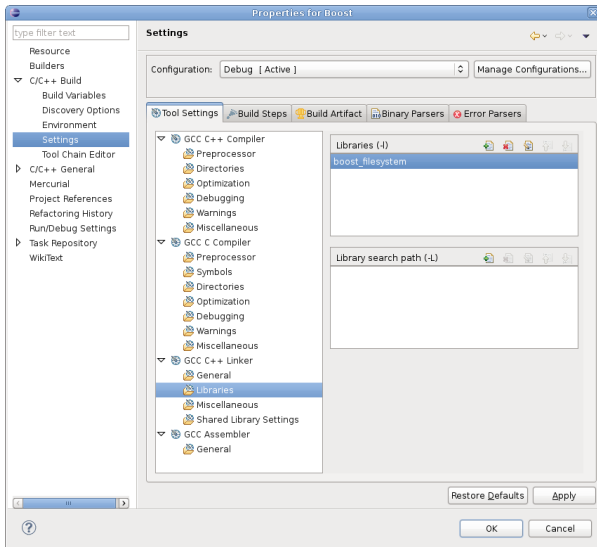
- Sind Sammlungen von vorkompilierten Object-Dateien die in das Programm gelinkt werden können
- Werden vom Linker benötigt um Referenzen auf Funktionen aufzulösen
- Befinden sich i. d. R. in `/usr/lib` und `/lib`

Ein Minimalbeispiel (Forts.)

- Referenzen können ohne die externe Bibliothek `libboost_filesystem.a` nicht aufgelöst werden
- Diese muss explizit gelinkt werden

```
$ g++ main.cpp -lboost_filesystem -o main
```
- `-lNAME` versucht die Bibliothek `libNAME.a` aus den Standard Bibliotheksverzeichnissen zu linken
- die Executable enthält anschließend auch den Maschinencode für die im Programm verwendeten Boost-Funktionen

Boost C++ Libraries



Smart Pointer

- Verhalten sich wie normale Zeiger
- Geben jedoch referenzierte Objekte automatisch frei sobald sie nicht mehr benötigt werden
- Erleichtern Operationen mit dynamisch alloziertem Speicher
- Vermeidung von Memoryleaks und Fehlern die durch schlechtes Speichermanagement entstehen

Smart Pointer in Boost

- Boost besitzt sechs verschiedene Typen:
 - ▶ `scoped_ptr<T>`
 - ▶ `scoped_array<T>`
 - ▶ `shared_ptr<T>`
 - ▶ `shared_array<T>`
 - ▶ `weak_ptr<T>`
 - ▶ `intrusive_ptr<T>`
- Der Operator `delete` wird bei ihrer Nutzung nicht benötigt

Ein Minimalbeispiel ohne Smart Pointer

Listing 3: main.cpp

```
1  #include "CSample.h"
2
3  int main() {
4
5      CSample *pSample = new CSample();
6      pSample->compute();
7
8      delete pSample;
9  }
```

Der Boost `scoped_ptr<T>`

- garantiert automatisches Löschen wenn der Zeiger den Gültigkeitsbereich verlässt
- der Operator `->` ist überladen und greift auf das gekapselte Objekt zu

Listing 4: main.cpp

```
1 #include <boost/scoped_ptr.hpp>
2 #include "CSample.h"
3
4 int main() {
5
6     boost::scoped_ptr<CSample> pSample(new CSample);
7     pSample->compute();
8
9 }
```

Der Boost `scoped_ptr<T>` (Forts.)

- Eignet sich besonders für die automatische Freigabe von lokalen Objekten
- Performanz wie bei normalen Zeigern

Problem

- Mehrere Zeiger auf das gleiche Objekt sind nicht möglich

Listing 5: Restriktionen bei `scoped_ptr<T>`

```
1  scoped_ptr<CSample> pA(new CSample);
2  scoped_ptr<CSample> pB = pA;    // Compiler-Fehler: nicht erlaubt
3  pA = new CSample;             // Compiler-Fehler: nicht kompatibel
```

Der Boost `scoped_ptr<T>` (Forts.)

- Folgendes ist aber möglich:

Listing 6: main.cpp

```
1 T* scoped_ptr<T>::get()           // gibt den enthaltenen Zeiger zurück
2   scoped_ptr<T>::reset(T *)       // ersetzt den enthaltenen Zeiger mit
3                                   // einer neuen Instanz. Die vorige
4                                   // Instanz wird dabei freigegeben
```

Der Boost `shared_ptr<T>`

- Ermöglicht es mehrere Smart Pointer auf das gleiche Objekt zeigen zu lassen
- Ist ein referenzgezählter Zeiger der überwacht wieviele Zeiger auf ein Objekt verweisen

Problem

- Keine zirkulären Referenzen möglich

Der Boost `shared_ptr<T>` (Forts.)

Listing 7: main.cpp

```
1  #include <iostream>
2  #include <boost/shared_ptr.hpp>
3  #include "CSample.h"
4
5  int main() {
6
7      boost::shared_ptr<CSample> p1(new CSample());
8      std::cout << "use count: " << p1.use_count() << std::endl;
9
10     boost::shared_ptr<CSample> p2 = p1;
11     std::cout << "use count: " << p1.use_count() << std::endl;
12
13     p1.reset();
14     std::cout << "use count: " << p2.use_count() << std::endl;
15 }
```