

# Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

21. Januar 2019

# Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

- 1 Einführung**
- 2 Wiederholung: Programmieren in C**
- 3 Einführung in C++**
- 4 Standard-Bibliotheken (STL)**
- 5 C++11**
- 6 Die Boost-Bibliothek**
  - 6.1** C++11-Erweiterungen für ältere Compiler
  - 6.2** Spracherweiterungen
  - 6.3** Zeitangaben
  - 6.4** Interprozeßkommunikation
  - 6.5** Asynchroner Input/Output
- 7 Plug-In-Architekturen**
  - 6.1** Shared-Libraries
  - 6.2** Design-Überlegungen



Änderungen  
vorbehalten

# 6 Die Boost-Bibliothek

Überblick: <http://dieboostcppbibliotheken.de/>

## 6.1 C++11-Erweiterungen für ältere Compiler

### C++11

```
for (auto &i : container)
[](int i){ cout << i << endl; }
#include <cstdint>
#include <random>
...
```

### Boost

```
BOOST_FOREACH (int &i, container)
cout << boost::lambda::_1 << "\n"
#include <boost/cstdint.hpp>
#include <boost/random.hpp>
...
```

# 6 Die Boost-Bibliothek

## 6.2 Spracherweiterungen

- „Duck Typing“:  
`boost::any`
- Coroutinen:  
kooperatives Multitasking (Alternative zu Threads)  
anhalten und Wert-Rückgabe über `push_type`-Objekt,  
aufrufen und Wert-Abfrage über `pull_type`-Objekt
- Parameter-Namen:  
`BOOST_PARAMETER_NAME()`, `BOOST_PARAMETER_FUNCTION()`

## 6 Die Boost-Bibliothek

### 6.3 Zeitangaben

- Kalender (Schaltjahr-Problematik usw.)
- formatierte Ein-/Ausgabe
- Zeitzonen
- Zeitmessungen
- Messung der Code-Ausführungsgeschwindigkeit
- betriebssystemunabhängiges Warten:  
`this_thread::sleep_for (chrono::milliseconds (137));`

## 6 Die Boost-Bibliothek

### 6.4 Interprozeßkommunikation

- Gemeinsamer Speicher: *Shared Memory*
- Kernel verwaltet benannte Speicherbereiche
- Existenz persistent, unabhängig von Prozessen
- Spezialfall von *Mapped Region*
- in C (Unix): `shm_open()`, `mmap()`, Bibliothek: `-lrt`
- in C++ (Boost): betriebssystemunabhängig
- ... außer, man verwendet MS-Windows-spezifischen Shared Memory, der beim Beenden des Prozesses automatisch entfernt wird

## 6 Die Boost-Bibliothek

### 6.4 Interprozeßkommunikation

- Verwalteter gemeinsamer Speicher: *Managed Shared Memory*
- Ein C++-`map`-Container verwaltet benannte Variable innerhalb des Shared Memory.
- erzeugen mit `construct<>`, verwenden mit `find<>`, beides mit `find_or_construct<>`
- wieder entfernen mit `destroy<>`, `destroy_ptr<>`
- Speicherüberlauf: `bad_alloc`-Exception
- spezielle Typen: `boost::interprocess::string` und Container (enthalten Zeiger → müssen auf Shared Memory verweisen)
- Synchronisation: *Mutex*-Variable `interprocess_mutex`, `named_mutex`

## 6 Die Boost-Bibliothek

### 6.5 Asynchroner Input/Output

- serielle Schnittstelle, TCP/IP, ...  
aber auch: Timer
- mit Callbacks  
statt mit blockendem I/O und `select()`
- Timer: Zeitpunkt, Wartezeit
- TCP/IP: Resolver, Verbindungsaufbau, Daten
- statt Schleife: Callback re-installiert sich selbst



## 7 Plug-In-Architekturen

Markus Ewald („Cygon“):

<http://blog.nuclex-games.com/tutorials/cxx/plugin-architecture/>

**Hinweis:** Implementation in C++ statt C erfordert, daß Programm- und Plug-In-Entwickler denselben Compiler in derselben Version verwenden.

**Hinweis:** Das Mischen von GNU-GPL-Code mit GNU-GPL-inkompatiblem Code ist in der beschriebenen Weise **nicht** zulässig, da das Plug-In nicht als separater Prozeß läuft.

Das Mischen von GNU-LGPL-Code mit GNU-LGPL-inkompatiblem Code ist in der beschriebenen Weise hingegen zulässig.

# 7 Plug-In-Architekturen

## 7.1 Shared Libraries

Sprache: C

- kleinster gemeinsamer Nenner
- konkret: Linker-Symbole

Unter Unix:

- Shared Library erzeugen: `gcc`-Option `-shared` beim Linken
- bei Programmstart linken: auf `gcc`-Kommandozeile mit angeben  
Environment-Variable: `LD_LIBRARY_PATH`
- zur Laufzeit linken: `#include <dlfcn.h>`, `dlopen()`

Unter MS-Windows:

- zur Laufzeit linken: `#include <windows.h>`, `LoadLibrary()`

# 7 Plug-In-Architekturen

## 7.2 Design-Überlegungen

Design-Idee 1:

- Plug-In stellt Factory-Funktion mit Parameter zur Verfügung

→ mehrere Plug-Ins: Typumwandlungen erforderlich

Design-Idee 2:

- Plug-In stellt Registrierungs- und Deregistrierungs-Funktionen zur Verfügung, um Methoden zu registrieren

→ Verwaltung mehrerer Implementationen ist schwierig

→ Funktionalität muß auf sehr niedriger Ebene bekannt sein

Design-Idee 3:

- Programm definiert Interface für Factory-Klasse für Subsysteme
- Plug-In-Manager erlaubt Registrierung von Factory-Implementationen

Design-Idee 4:

- jedes Subsystem erlaubt Plug-Ins die Registrierung von Implementationen

# Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

- 1 Einführung**
- 2 Wiederholung: Programmieren in C**
- 3 Einführung in C++**
- 4 Standard-Bibliotheken (STL)**
- 5 C++11**
- 6 Die Boost-Bibliothek**
- 7 Plug-In-Architekturen**
  - 6.1 Shared-Libraries**
  - 6.2 Design-Überlegungen**