Fachhochschule Nordwestschweiz
Hochschule für Technik

# Programmieren in C++

Christian Lang (Lac)

15. November 2019

# Erfahrungen aus der Praxis

# Inhalt

- Implementation by Example: Observer-Pattern
- OpenSource und Arbeitsumfeld
- Embedded Systems
- Concurrency vs. Parallelism
- Event-Loop
- Beispiel: sd-event
- Empfehlungen

# Fragestellung: Wie implementiere ich das Observer-Pattern?

- Problem ist bekannt $\rightarrow$ hat sogar einen Namen
- Gibt es eine Implementation oder Helper in der Standard-Library?
- Performance-Anforderungen?

# 1. Ansatz: Smart-Pointer und Funktionsobjekte

```
1  // Callback-Funktion
2  using Callback = std::function<void(int new_value)>;
3
4  // Callback-Handler
5  std::weak_ptr<Callback> subscriber_;
```

- Maximale Freiheit für `subscriber_`
- sehr Implizit beim Implementieren
- `std::weak_ptr` damit Zerstörung des Observers nicht verhindert wird

# 2. Ansatz: Smart-Pointer und Interface

```
1  struct CallbackInterface {
2    virtual ~CallbackInterface() = default;
3    virtual void Notify(int new_value) = 0;
4  }
5
6  struct ExampleManager {
7    void Subscribe(std::weak_ptr<CallbackInterface> subscriber) {
8      subscriber_ = subscriber;
9    }
10   void OnChange(int new_value) {
11     if (const auto subs = subscriber_.lock()) {
12       subs->Notify(new_value);
13     }
14   }
15
16  private:
17   std::weak_ptr<CallbackInterface> subscriber_;
18 }
```

# 3. Ansatz: Libraries suchen

- Konzept abklären: Wikipedia
- Observer-Pattern hat unterschiedliche Namen:
  - Signal-Slot (Qt)
  - Callback
- Google/Stack-Overflow
- → nano-signal-slot

```
1  struct CallbackInterface : public Nano::Observer {
2    virtual ~CallbackInterface() = default;
3    virtual void Notify(int new_value) = 0;
4  }
```

- automatisches Abmeldung bei Zerstörung: `Nano::Observer`
- mehrere Slots pro Signal

# Implementation mit Nano-Signal-Slot

```
1  struct ExampleManager {
2    void Subscribe(std::weak_ptr<CallbackInterface> subscriber) {
3      if (const auto subs = subscriber.lock()) {
4        Subscribe(subs.get());
5      }
6    }
7    void Subscribe(CallbackInterface* subscriber) {
8      signal_.connect<CallbackInterface, &CallbackInterface::Notify>(subscriber);
9    }
10
11   void OnChange(int new_value) {
12     signal_.emit(new_value);
13   }
14
15  private:
16   Nano::Signal<void(int)> signal_;
17 }
```

# OpenSource

- C++ ist ohne riesiges Environment aus Libraries konzipiert
  - obwohl die Standard-Library aktuell sehr stark wächst
  - bietet mehr Freiheiten für Optimierungen/Plattformen/etc.
- viele kleinere Libraries (meist Header-Only) auf Github
  - nano-signal-slot
  - units
  - cpp-utilities
  - protocol-buffers
  - libdivide
- diverse grössere etablierte Libraries
  - abseil: Moderne C++ Features für C++11
  - boost: Stark Template-isiert
  - POCO: Einfach gehalten. Orientiert sich an Java
- C-Libraries auch verwenden (vorallem auf System/Linux Ebene)

# Arbeitsumfeld

- Continuous Integration (Unit-/Integration-/System-Tests, etc.)
  - Jenkins
  - Bamboo
  - GitLab-CI
- Testing-Frameworks
  - googletest+googlemock
  - Catch2
- Code-Style
  - Google C++ Style Guide
  - CppCoreGuidelines
- Code-Style/Check-Tools → Forciert in Build
  - cpplint
  - cppcheck
  - cppclean
  - clang-tidy

# Embedded Systems

- heutzutage sehr breit verwendeter Begriff
- kann folgende Aspekte enthalten:
  - Runtime Plattform hat (stark) begrenzte Ressourcen (CPU-Cores, CPU-Takt, RAM, Harddisk, etc.)
  - Runtime Plattform andere Architektur als Entwicklungsplattform (x86, ARM, $\mu$C)
  - in ein Produkt integriert und von aussen nicht direkt erkennbar/ansprechbar
  - beinhaltet Aufgaben-spezifische Hardware
  - heutzutage meist vernetzt (IoT)
  - eigenes Betriebssystem $\rightarrow$ Base-Support-Package (BSP)
- Anwendungs-Beispiele:
  - Waschmaschine
  - Funksysteme
  - Verbrennungs-Motor-Steuerung
  - Notfallsysteme

# Tools für Embedded-Entwicklung

- passende Programmiersprache
  - C/C++
  - generierter C-Code z.B. aus Matlab
  - Rust
- Compilation
  - andere Plattform → Cross-Compilation
  - nicht auf Host ausführbar
- Testing/Debugging
  - Emulatoren (Qemu) oder Virtuelle Maschinen (VMs)
  - Hardware-Zugriffe abstrahieren
  - portabel programmieren → Unit-Testing auf Host möglich
- BSP
  - PTXdist
  - Buildroot
  - Yocto

# Concurrency vs. Parallelism

| Concurrency | Parallelism |
| --- | --- |
| gefühlte Parallelität<br>→ benötigt nur 1 CPU-Core | benötigt mehrere CPU-Cores |

**Verwendung**

- Concurrency: zeitlich unabhängige Tasks
- Parallelism: Problem schneller lösen

# Implementation von Concurrency

- Architektur einfach halten:
  - 1 Thread pro Prozess
  - Probleme in Prozesse aufteilen
  - Interprocess-Communication (IPC)

Anstatt mehrere Threads für Concurrency: → Event-Loop

- Definition: Wikipedia: Event loop
- kooperatives Scheduling (userspace)
- baut meist auf Betriebssystem-Funktionen auf
- diverse Libraries
  - WinMain (C++)
  - GLib Main Event Loop (C)
  - sd-event (C)
  - Boost.Asio (C++)

# Beispiel: sd-event

```
1  #include <cassert>
2
3  #include <sys/socket.h>
4
5  #include <systemd/sd-event.h>
6  #include <systemd/sd-daemon.h>
7
8  // create event loop
9  sd_event* event_loop = nullptr;
10 int result = sd_event_default(&event_loop);
11 assert(result >= 0);
12
13 // enable watchdog handling
14 result = sd_event_set_watchdog(event_loop, true);
15 assert(result >= 0);
```

# Beispiel: sd-event

```
1  static int io_handler(sd_event_source* es,
2      int fd, uint32_t revents, void* userdata) {
3    // ...
4    return 0;
5  }
6
7  // open UDP socket with asynchronous operation mode
8  int fd = socket(AF_INET, SOCK_DGRAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0);
9  assert(fd >= 0);
10 // ...
11
12 // register I/O event for incoming messages on fd
13 sd_event_source* io_event = nullptr;
14 result = sd_event_add_io(event_loop, &io_event, fd, EPOLLIN, io_handler, nullptr);
15 assert(result >= 0);
```

# Beispiel: sd-event

```
1  static int timer_handler(sd_event_source* s,
2      uint64_t usec, void* userdata) {
3    // ...
4    return 0;
5  }
6
7  // register timer event with interval 1ms (accuracy 100us)
8  sd_event_source* timer_event = nullptr;
9  result = sd_event_add_time(event_loop, &timer_event, CLOCK_MONOTONIC,
10                             1000, 100, timer_handler, nullptr);
11 assert(result >= 0);
12
13 // notify system manager about finished startup
14 sd_notify(false, "READY=1\nSTATUS=Startup completed.");
15
16 // loop forever and process events
17 result = sd_event_loop(event_loop);
18 assert(result >= 0);
```

# Mapping von C-Callbacks auf C++

```
1  class TimerHandler {
2   public:
3    explicit TimerHandler(sd_event* event_loop) {
4      assert(sd_event_add_time(event_loop, &timer_event_, CLOCK_MONOTONIC,
5                               1000, 100, Handler, nullptr) >= 0);
6    }
7
8   private:
9    static int Handler(sd_event_source* s, uint64_t us, void* userdata) {
10     const auto instance = static_cast<TimerHandler*>(userdata);
11     return instance->HandlerImpl(s, std::chrono::microseconds(us));
12   }
13   int HandlerImpl(sd_event_source* s, const std::chrono::microseconds& ts) {
14     // ...
15     return 0;
16   }
17
18   sd_event_source* timer_event_;
```

# Empfehlungen

- Wissen: Bücher von Scott Meiers
- Alle Tools/Libraries/etc. aus diesem Modul