

# Verteilte Systeme

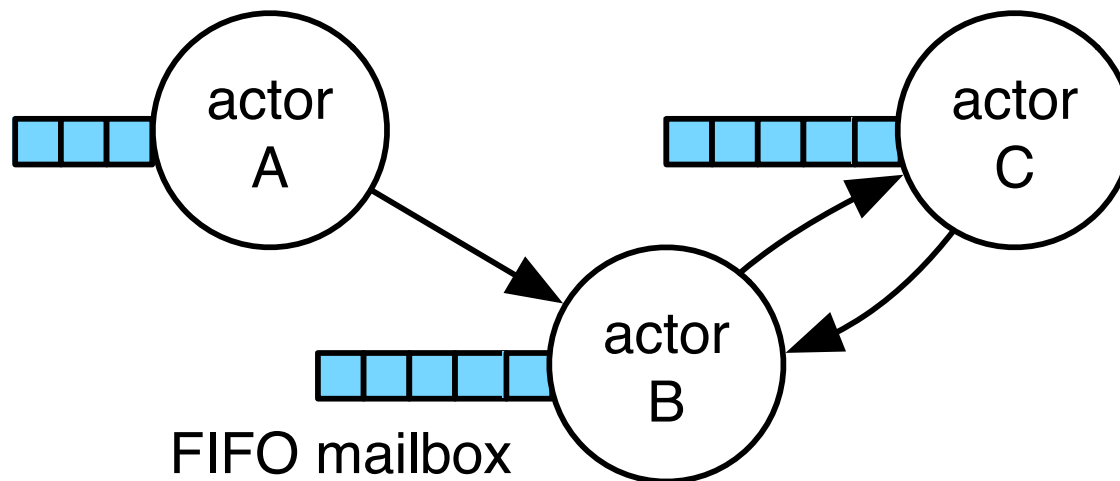
Programmieren im  
Aktormodell mit C++

# Aktoren in Verteilten Systemen

- ◆ Relevant in hochskalierbaren, reaktiven Systemen
  - Web-Dienste, Datenbanken, IoT-Anwendungen, Kommunikationsdienste wie z.B. WhatsApp, ...
  - Microservice-Architekturen
- ◆ Für die Praxis wichtig:
  - Message Passing als Entwurfsmuster
  - Fehlerbehandlung in hochverfügbaren Diensten
  - Hochstehende Abstraktionen mit effizienter Laufzeit

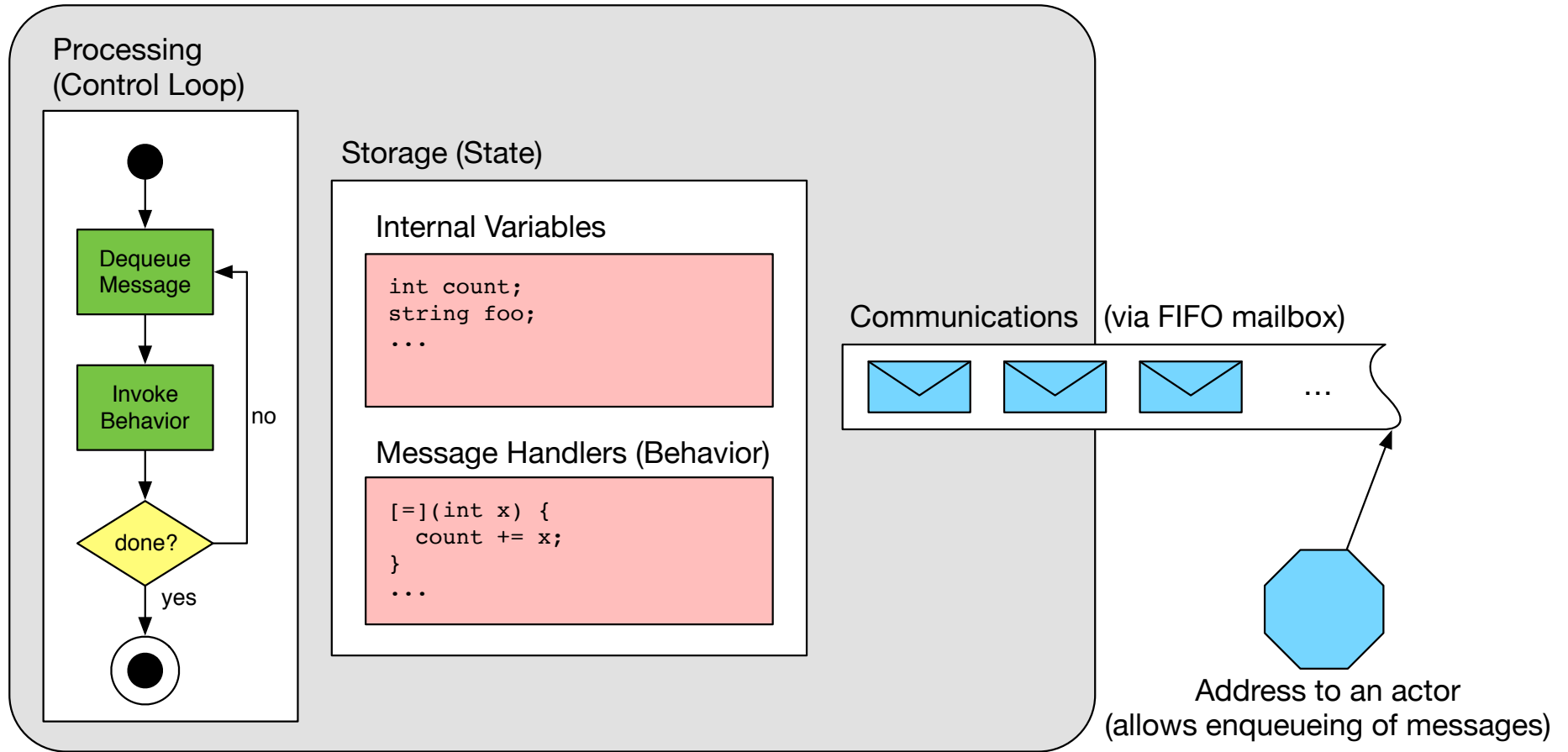
# Das Aktormodell

- ◆ Aktoren kapseln „Processing, Storage & Communications“
- ◆ Asynchroner Nachrichtenaustausch
- ◆ Keine geteilten Speicherbereiche
- ◆ Hierarchische Fehlerbehandlung



# Anatomie eines Aktors

Actor



# Orchestrierung von Aktoren

- ◆ Jedes Individuum agiert gemäß eines Skriptes
- ◆ Aktoren sind Agenten mit Zielen und Verhalten
- ◆ Ein Programm ist eine Choreographie vieler Aktoren
- ◆ „Actor Model“ entspringt einer „Theateraufführung“-Metapher



# Hauptmerkmale von Aktoren

- ◆ Aktoren sind inhärent nebenläufig
  - Parallel ausführbar, da sie keine Speicherbereiche teilen
  - Einziger Synchronisationspunkt ist die Mailbox
- ◆ Kommunikation zwischen Aktoren ist netzwerktransparent
  - Auf Quellcode-Ebene nicht ersichtlich ob Komm. lokal ist
  - Entferntes Instanzieren ändert Anwendungslogik nicht
- ◆ Leichtgewichtige Aktoren skalieren besser als Threads\*
  - Wenige hundert Bytes RAM statt mehrere tausend
  - Keine Verwaltung durch das Betriebssystem erforderlich

# Programmiermodell

- ◆ Aktoren sind nachrichtengetrieben (reaktiv)
- ◆ In Reaktion auf eine Nachricht kann ein Aktor:
  1. Nachrichten senden
  2. Neue Aktoren starten
  3. Bestimmen wie die nächste Nachricht verarbeitet wird

# Fehlerbehandlung

- ◆ Fehler haben keine Seiteneffekte auf andere Aktoren
  - Keine Propagierung durch Seitenkanäle wie Exceptions
- ◆ Explizites Monitoring/Linking zum behandeln entfernter Fehler
  - Fehler sind lokal, aber entfernt erkenn- und behandelbar
- ◆ Monitoring: unidirektional, signalisiert mit „down“ Nachrichten
  - Erlaubt „stilles Beobachten“ von Fehlern entfernter Aktoren
- ◆ Linking: bidirektional, signalisiert mit „exit“ Nachrichten
  - Erlaubt es Lebenszeit von Aktoren zu verknüpfen



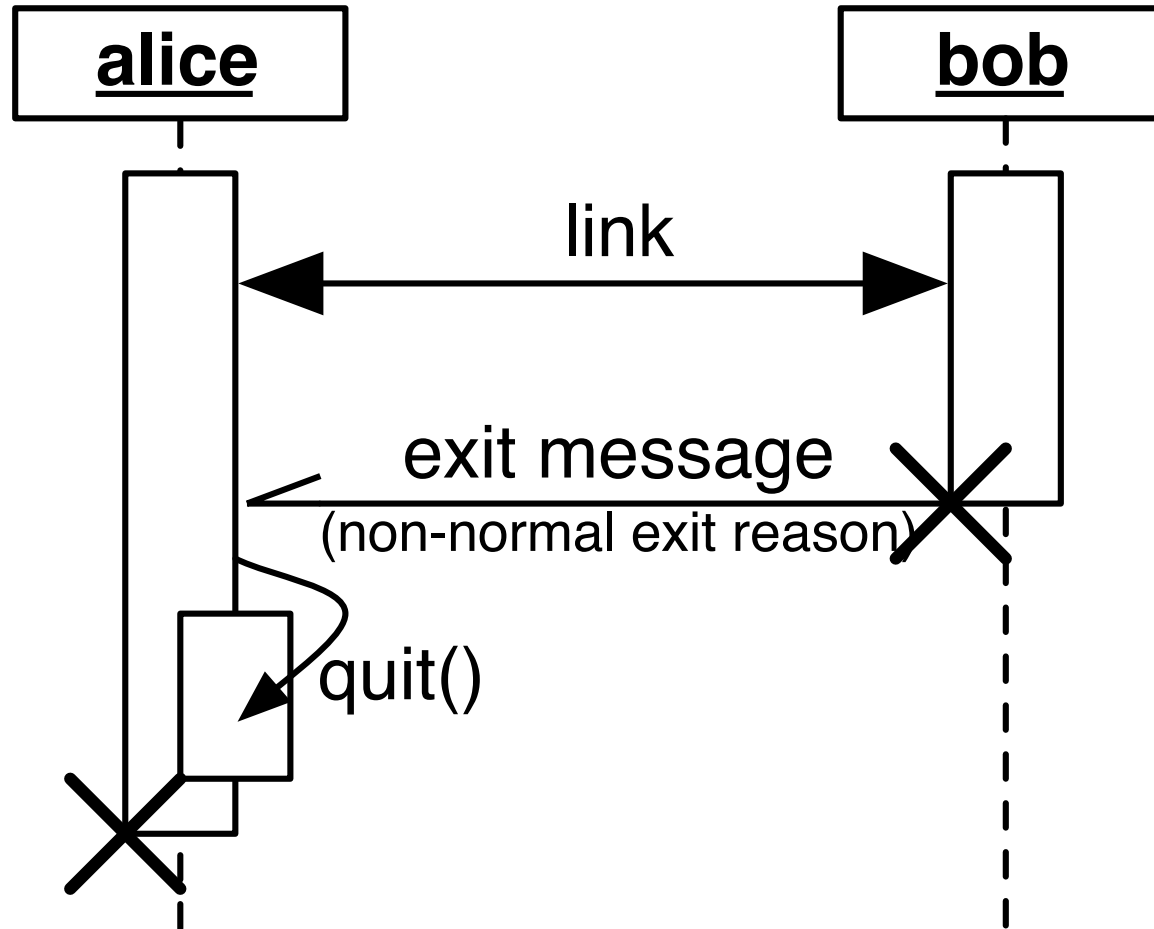
# Monitoring

- ◆ Fehlerbehandlung bei lose/temporär gekoppelten Aktoren
  - Z.B.: Clients können auf Serverausfall reagieren
  - Erlaubt Fallback-Strategien auf Client-Ebene
- ◆ Vergleich zu Links:
  - Clients beobachten Server-Lebenszeit unidirektional
  - Server nicht benachrichtigt bei Client-Ausfällen
  - Keine Standardstrategien bei „down“ Nachrichten

# Linking

- ◆ Linking koppelt die Lebenszeit von Aktoren
  - Fällt ein Aktor mit einem Fehler aus beendet er seine Links
  - Linking erlaubt „alle leben oder keiner“ Semantik
  - Ausnahme: Supervisor behandeln „exits“ manuell
  - Bei Supervision: Worker sollen mit Supervisor ausfallen
- ◆ Supervisor erlauben dynamisches Re-Deployment
  - Ausgefallene Worker werden ersetzt
  - Worker können auf anderen Knoten neugestartet werden

# Standardverhalten von Links



# CAF – Das „C++ Actor Framework“

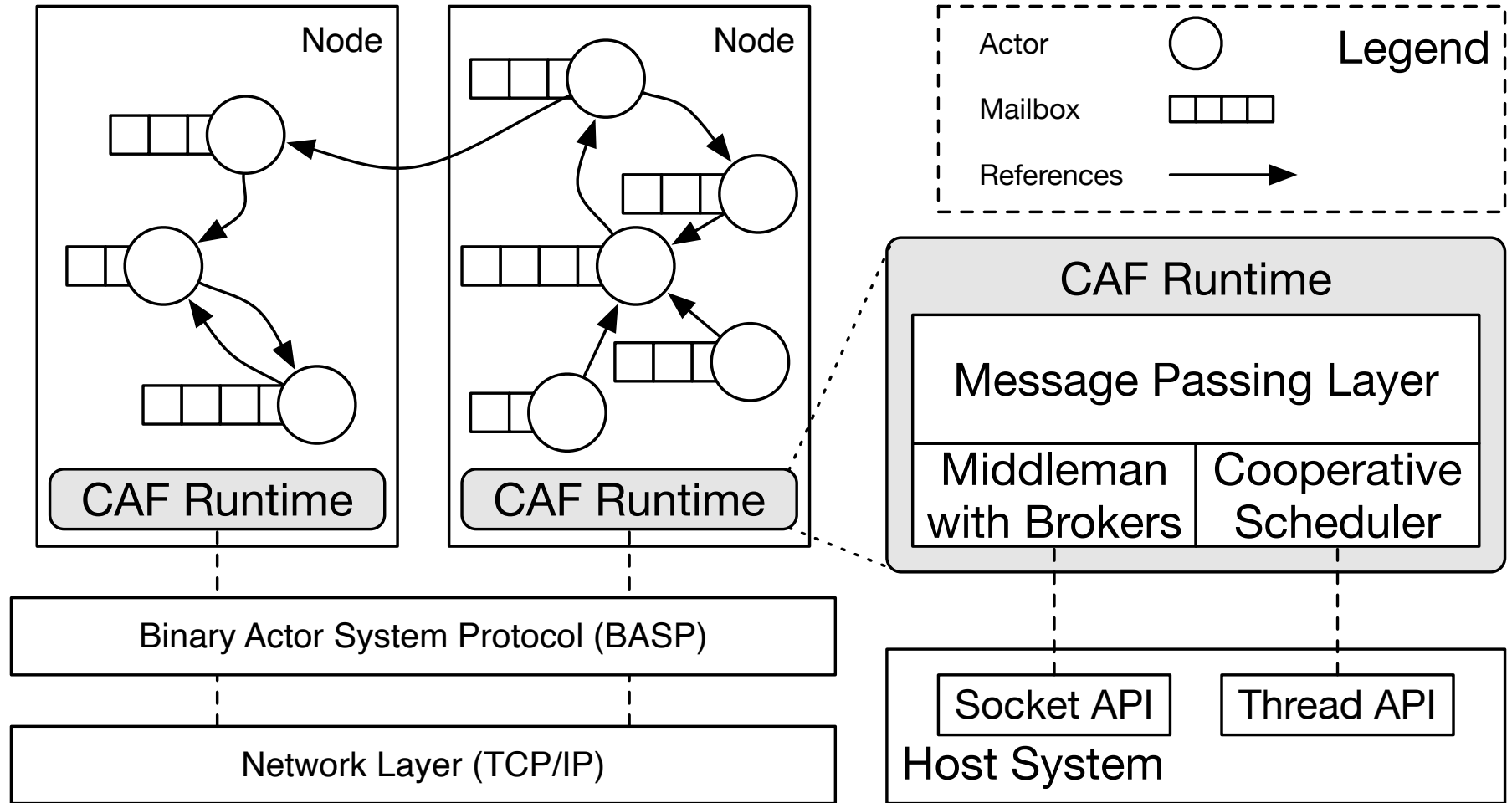
- ◆ Leichtgewichtige Implementierung des Aktormodells
- ◆ Aktiv entwickelt seit 2011 in der INET Arbeitsgruppe
- ◆ Open Source Software mit BSD Lizenz
- ◆ Aktive, internationale Community
- ◆ Verfügbar auf GitHub:

<https://github.com/actor-framework/actor-framework>

# CAF Zusammengefasst

- ◆ Liefert Bausteine für Infrastruktur-Software
  - Web-Services, Kommunikations- oder MMO-Backends
  - Hohe Anforderungen an Elastizität & Performance
- ◆ Legt den Fokus auf Robustheit & Effizienz
  - Robust gegenüber Ausfall einzelner Aktoren/Systemen
  - Effizient in Speicherverbrauch und Laufzeitverhalten
- ◆ Relevant für Anwender aus Industrie und Forschung
  - Hohe Abstraktionsschicht
  - Native Laufzeitumgebung (C++)

# Architektur von CAF



# Grundlagen C++

- ◆ C++ ist *nicht* „C mit Klassen“ und *nicht* „Java ohne GC“
  - Stärkeres Typsystem als C
  - Keine Virtualisierung / GC wie in Java
- ◆ Lebendige Sprache mit derzeit 3-jährigem Release-Cycle
  - Achtung: viele Bücher und Online-Quellen veraltet!
  - Großer Einfluss funktionaler Konzepte, z.B. Lambdas
- ◆ Alle Beispiele aus dieser Vorlesung sind online verfügbar:
  - <https://github.com/inetrg/vs-cpp>

# Minimales C++ Programm

```
#include <iostream>
```

Standard Header für  
I/O-Ströme

```
int main(int, char**) {  
    std::cout << "Hello World"  
    << std::endl;  
}
```

Hauptfunktion eines  
C++ Programms

std ist ein  
Namensraum, cout ist  
das Standardausgaben-  
Singleton, << ist der  
Ausgabestrom-Operator

std::endl schreibt  
einen Zeilenumbruch  
und leert Schreibpuffer;  
alternativ: "Hello  
World\n"



## Variante #2: Individueller Import

```
#include <iostream>

using std::cout;
using std::endl;

int main(int, char**) {
    cout << "Hello World" << endl;
}
```

Importiert *einzelne* Namen aus dem Namensraum `std` in den aktuellen Scope (hier: globaler Namensraum / ganze Datei)

## Variante #3: Importieren von std

```
#include <iostream>

using namespace std;

int main(int, char**) {
    cout << "Hello World" << endl;
}
```

Importiert *alle* Namen aus dem Namensraum std in den aktuellen Scope (hier: globaler Namensraum / ganze Datei); kann Mehrdeutigkeiten erzeugen, insbesondere in Header-Dateien

## Variante #3b: Importieren von std

```
#include <iostream>
```

```
int main(int, char**) {  
    using namespace std;  
    cout << "Hello World" << endl;  
}
```

Import gilt nur innerhalb  
der main Funktion

# Dynamische Arrays: `vector<T>`

```
std::vector<int> xs{10, 20, 30};  
xs.emplace_back(40);  
for (int x : xs)  
    std::cout << x << ' ';  
std::cout << std::endl;
```

Dynamisches Array  
initialisiert als Intervall  
[10, 20, 30]

Fügt 40 am Ende des  
Arrays ein

Schreibt für jedes `x` in  
`xs` den Wert auf die  
Standardausgabe  
(Konsole)

# Scoping: Beispiel #1

```
class scoped {  
public:  
    scoped() { std::cout << "scoped() \n"; }  
    ~scoped() { std::cout << "~scoped() \n"; }  
};  
  
int main(int, char**) {  
    std::cout << "enter main\n";  
    scoped x;  
    std::cout << "leave main\n";  
}
```

Standard-Konstruktor  
(keine Argumente)

Destruktor (aufgerufen  
sobald das Objekt  
zerstört wird)

Programmausgabe (x  
wird beim Verlassen  
von main zerstört):  
enter main  
scoped()  
leave main()  
~scoped()

## Scoping: Beispiel #2

```
class scoped {  
public:  
    scoped() { std::cout << "scoped() \n"; }  
    ~scoped() { std::cout << "~scoped() \n"; }  
};  
  
int main(int, char**) {  
    std::cout << "enter main\n";  
    { scoped x; }  
    std::cout << "leave main\n";  
}
```

Programmausgabe (x lebt nicht mehr im Scope von main sondern in Teil-Scope):

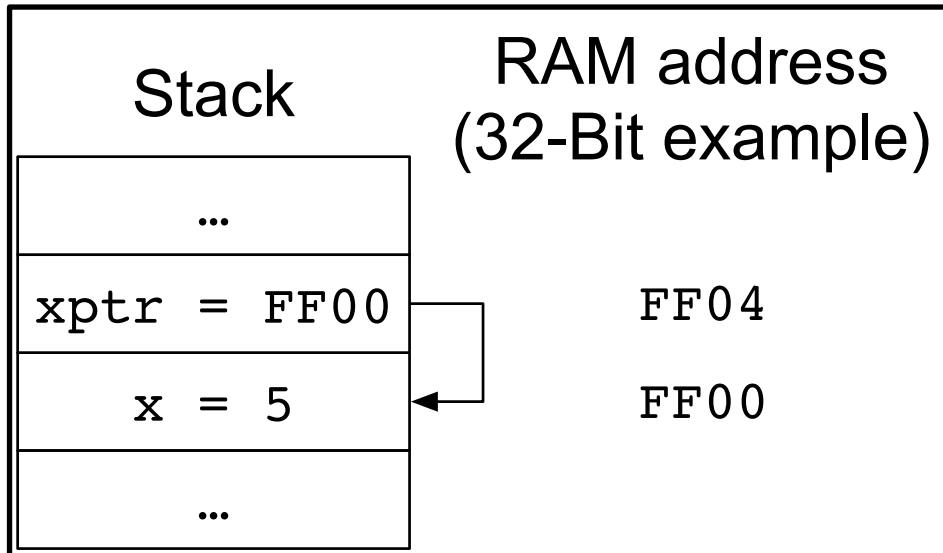
```
enter main  
scoped()  
~scoped()  
leave main()
```

# Zeiger auf Stack-Objekte

```
int x = 5;
int* xptr = &x;
*xptr = 10;
cout << "x = "
      << x << endl;
```

`int*` ist ein Zeiger auf einen `int`, `&x` gibt die Adresse von `x` zurück

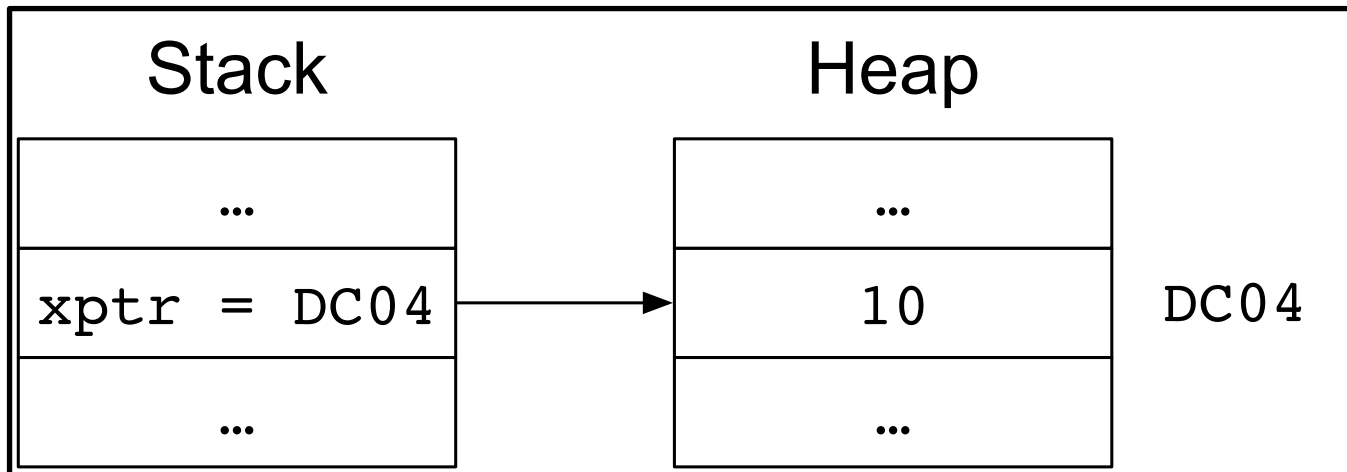
`*xptr` *dereferenziert* den Zeiger, d.h. greift auf den Inhalt der gespeicherten Adresse zu und modifiziert somit indirekt den Wert von `x`



# Sichere Zeiger auf Heap-Objekte

```
std::unique_ptr<int> xptr;  
assert(xptr == nullptr);  
xptr.reset(new int(10));  
assert(xptr != nullptr);  
cout << "x = " << *xptr << endl;
```

unique\_ptr ist ein intelligenter Zeiger (Smart Pointer), der Speicher „besitzt“ und wieder freigibt wenn der Scope verlassen oder reset aufgerufen wird





# Referenzen

```
int x = 5;  
int& xref = x;  
int* xptr = &xref;  
xref = 10;  
int& xref2 = *xptr;  
xref2 = 20;  
cout << "x = " << x << endl;
```

Vollwertiges Alias für x

Die Adresse einer Referenz ist die Adresse des referenzierten Objektes

Zeiger geben eine Referenz zurück wenn sie dereferenziert werden

# Typinferenz

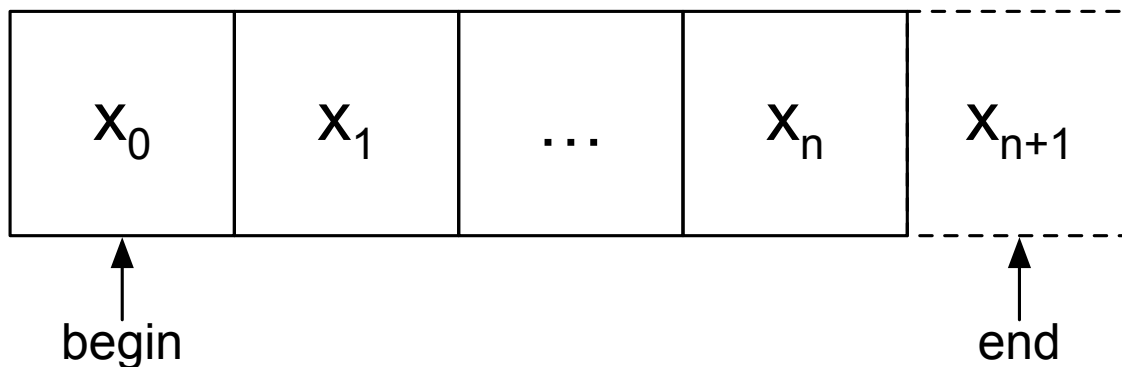
- ◆ Typinferenz leitet den Typen von der Initialisierung ab
  - Manuelle Typzuweisung redundant
  - Typen sind in vielen Fällen lang oder schlicht unbekannt
- ◆ Syntax: `auto <var> = <expr>;`
  - `auto`: Platzhalter für den tatsächlichen Typen
  - `<var>`: Name der Variablen
  - `<expr>`: Initialisierung der Variable
- ◆ Beispiel:
  - `auto x = make_object();`

# Typinferenz mit Referenzen

- ◆ `auto` kann mit Typmodifikatoren ergänzt werden
  - `auto&` ist eine Referenz auf den abgeleiteten Typen
  - `auto*` ist ein Zeiger auf den abgeleiteten Typen
  - `const auto&` ist eine unveränderliche Referenz
  - `const auto` ist ein unveränderlicher Wert
- ◆ Referenzen auf abgeleitete Typen insb. beim Iterieren sinnvoll
  - Vermeidet überflüssige Kopien durch Referenzen
  - Hält Quelltext kurz und prägnant durch Typinferenz

# Iteratoren

- ◆ Positionszeiger für Datenstrukturen:
  - $*i$ : Zugriff auf das Element an Position  $i$
  - $++i$ : Bewegt den Iterator zur nächsten Position
- ◆ Jede Datenstruktur  $T$  ist modelliert als Intervall  $[x_0, \dots, x_{n+1})$ 
  - $T::begin()$ : Iterator **auf** das erste Element
  - $T::end()$ : Iterator **hinter** das letzte Element



# for Schleife

- ◆ Klassische Schleife: `for (<init>; <guard>; <step>)`
  - `<init>` deklariert und initiiert die Schleifen-Variable
  - `<guard>` spezifiziert Abbruchkriterium
  - `<step>` weist Schleifen-Variable nächsten Wert zu
- ◆ Beispiele:
  - `for (int x = 0; x < 5; ++x):` Iteriert [0, 1, 2, 3, 4]
  - `for (int x = 0; x < 5; x += 2):` Iteriert [0, 2, 4]
  - `for (auto i=xs.begin(); i!=xs.end(); ++i):` Iteriert `xs`
  - `for (;;):` Endlosschleife (alle drei Teile sind optional)

## for-each Schleife

- ◆ Kompakte Schleife: `for (<var> : <xs>)`
  - `<var>` deklariert die Schleifen-Variable
  - `<xs>` Datentyp mit `begin()` und `end()`
  - Iteriert immer über alle Element in `<xs>`, äquivalent zu:  

```
for (auto i = <xs>.begin(); i != <xs>.end(); ++i) {  
  <var> = *i; ... }
```
- ◆ Beispiele:
  - `for (auto& x : xs)`: Iteriert alle Element als Referenz
  - `for (auto x : {1, 2, 4})`: Iteriert Intervall [1, 2, 4]

# while Schleife

- ◆ Schleife mit Laufbedingung: `while ( <condition> )`
  - `<condition>` boolesche Ausführungsbedingung
  - Schleife bricht ab sobald `<condition> == false`
- ◆ Beispiele:
  - `int i = 4; while (--i > 0) {}`: Iteriert [3, 2, 1]
  - `int i = 4; while (i-- > 0) {}`: Iteriert [3, 2, 1, 0]
  - `while (true)`: Endlosschleife

# Funktionen

- ◆ **Freie Funktionen:** an Namensräume gebunden
  - `int f();`
  - `namespace foo { void bar(int); }`
- ◆ **Member-Funktionen (Methoden):** an Klassen gebunden
  - `struct s1 { void f(int); };`
  - `struct s2 { static void g(int); };`
- ◆ **Anonyme Funktionen (Lambdas):** an Variablen gebunden
  - `auto f = [](int x) { return x * 2; };`
  - `auto g = [y](int x) { return x * y; };`



# Lambda Ausdrücke

- ◆ Struktur:  $[\langle \text{captures} \rangle](\langle \text{args} \rangle) \rightarrow \langle \text{res} \rangle \{ \dots \}$ 
  - $\langle \text{captures} \rangle$ : Kopien/Referenzen auf Scope-Variablen
    - $[ ]$ : Kein Zugriff auf äußeren Scope (stateless lambda)
    - $[ = ]$ : Zugriff auf alle Variablen als Kopie
    - $[ \& ]$ : Zugriff auf alle Variablen als Referenz
    - $[ \&, x ]$ : Zugriff auf  $x$  als Kopie, sonst Referenz
    - $[ =, \&x ]$ : Zugriff auf  $x$  als Referenz, sonst Kopie
  - $\langle \text{args} \rangle$ : Argumente der anonymen Funktion
  - $\langle \text{res} \rangle$ : Rückgabewert (idR nicht explizit erforderlich)

# Lambdas als Prädikate

```
vector<string> names
{"Tom", "Tim", "Bart", "Harry"};
auto end = names.end();
auto i = find_if(names.begin(), end,
                 [](const string& name) {
                     return name.size() > 3;
                 });

if (i == end)
    cout << "Only short names.\n";
else
    cout << "First long name: " << *i << ".\n";
```

find\_if gibt den  
ersten Iterator zurück  
für den das Prädikat  
gilt, sonst end

# Lambdas mit State als Prädikate

```
vector<string> names
{"Tom", "Tim", "Bart", "Harry"};
set<string> blocklist{"Bart"};
if (any_of(names.begin(), names.end(),
           [&](const string& name) {
               return blocklist.count(name) > 0;
           }))
    cout << "Blocked name found!\n";
else
    cout << "All names are good to go!\n";
```

any\_of gibt **true** zurück wenn das Prädikat für mind. eines der Elemente gilt

# Prototyp und Definition

```
struct foo {  
    // prototype (foo.hpp file)  
    void bar();  
};  
// definition (foo.cpp file)  
void foo::bar() {  
    cout << "foo::bar()\n";  
}  
// usage  
foo x;  
x.bar();
```

Der Prototyp deklariert Name und Signatur einer Funktion in einem Header und erlaubt Import durch andere Komponenten

Die Definition befindet sich in einer separaten Datei und wird vom Compiler nur ein Mal übersetzt

# Überladen von Operatoren

- ◆ Nur vordefinierte Operatoren können überladen werden
- ◆ Operatoren existieren als freie und Member-Funktionen
- ◆ Beispiele für unäre (Member-) Operatoren:

`=, +=, -=, %=, *=, ^=, &=, |=, !, ++, --, ^, ~, &, *`

- ◆ Beispiele für (i.d.R. freie) binäre Operatoren:

`==, !=, <, >, <=, >=, +, -, *, /, <<, >>, ||, &&, ,`

- ◆ Member-Operatoren mit variabler Anzahl an Argumenten:

`[ ], ()`

## Eigene Typen ausgeben mit <<

```
struct point2d { int x; int y; };
```

Basistyp für alle I/O  
Ströme (z.B. cout)

```
std::ostream& operator<<(std::ostream& out,  
                        const point2d& x) {  
    return out << "point2d{" << x.x << ", "  
                << x.y << "}";  
}  
  
int main(int, char**) {  
    point2d p1{10, 20};  
    cout << p1 << endl;  
}
```

# Ownership-Transfer

- ◆ Klassen wie `vector` und `unique_ptr` **besitzen** Speicher
  - **Intern verwaltet** während der Lebenszeit des Objekts
  - **Freigeben beim zerstören** des Objekts
- ◆ **Speicher kann übertragen werden** mit `std::move(x)`
  - Ursprünglich besitzendes Objekt anschließend uninitialisiert
  - Besitzübertragende Referenzen haben die Form `T&&`
- ◆ Typen wie `unique_ptr` sind *move-only*-Typen
  - Können nicht kopiert werden
  - Modellieren Lebenszeit für Heap-allokierte Objekte

## std::move

```
unique_ptr<int> x;  
unique_ptr<int> y{new int(42)};  
auto print = [&] {  
    cout << "x = " << x << ", "  
         << "y = " << y << "\n";  
};  
print();  
// x = y => compiler error  
x = std::move(y); // ownership transfer  
print();
```

y besitzt den Speicher,  
der durch `new int`  
allokiert wurde

Nach dem Ownership-  
Transfer ist y  
uninitialisiert und x  
besitzt den allokierten  
Speicher



# Eigene Klassen

- ◆ `struct` und `class` sind austauschbar, einziger Unterschied:
  - Member einer `struct` sind standardmäßig `public`
  - Member einer `class` sind standardmäßig `private`  
(Achtung bei Vererbung mit `private` Basisklassen! → keine *is-a* Beziehung)
- ◆ Es gibt fünf Standardoperationen die ein Typ `T` anbieten *kann*
  1. `T ( )`: Standard-Konstruktor
  2. `T ( T&& )`: Move-Konstruktor
  3. `T ( const T& )`: Copy-Konstruktor
  4. `T& operator= ( T&& )`: Move-Zuweisung
  5. `T& operator= ( const T& )`: Copy-Zuweisung

# Virtuelle Methoden

- ◆ Methoden sind *nicht* überschreibbar per Default
  - `virtual` markiert Methoden als überschreibbar
  - Abstrakte Methoden werden mit `= 0` deklariert
- ◆ Basistypen müssen ihren Destruktor als `virtual` deklarieren
  - Nur virtuelle Destruktoren delegieren zum richtigen Typ
- ◆ Beispiel für einen abstrakten Basistypen:

```
struct base { virtual ~base();  
              virtual void f() = 0; };  
struct derived : base { void f() override; };
```

# Richtlinien für C++

- ◆ Kein `new` ohne Ownership-transfer in einen Smart Pointer
- ◆ Kein `delete` in Anwendungscode
- ◆ Klare Zuweisung von Ownership: Scoping und Smart Pointer
- ◆ Code soll *Intentionen* ausdrücken
- ◆ Bevorzuge Werte-Semantik wenn möglich
- ◆ Bevorzuge Funktionen ohne Seiteneffekte
- ◆ Nutze Algorithmen aus `<algorithm>` statt Schleifen
- ◆ Online-Übersicht zu C++ und zur Standardbibliothek:  
<https://cppreference.com>

# Grundlagen CAF

- ◆ CAF besteht aus vielen Teilkomponenten, z.B.:
  - Nebenläufigkeit mit Aktoren
  - Verteilung und netzwerktransparente Kommunikation
  - Publish/Subscribe Kommunikation für lose Kopplung
  - Konfiguration und Erweiterung (z.B. eigene Typen)
- ◆ Funktionaler Stil:
  - Implementieren von Aktoren als Funktionen
  - Separierung von Daten und Logik

# Minimale CAF Anwendung

```
#include "caf/all.hpp"  
#include "caf/io/all.hpp"
```

Importiert alle Klassen  
und Funktionen aus  
CAF

```
using namespace caf;
```

Ersetzt die reguläre  
Hauptfunktion in CAF  
Anwendungen

```
void caf_main(actor_system& sys) {  
    // ...  
}
```

actor\_system ist die  
Laufzeitumgebung für  
Aktoren

```
CAF_MAIN(io::middleman)
```

Anwendungs-  
Grundgerüst; lädt die  
Netzwerk-Komponente  
(io::middleman) und  
ruft caf\_main auf

# Konfigurierbare CAF Anwendung

```
struct config
: actor_system_config {
std::string msg = "Hello";
config() {
opt_group{custom_options_,
"global"}
.add(msg, "message,m",
"set output");
}
};

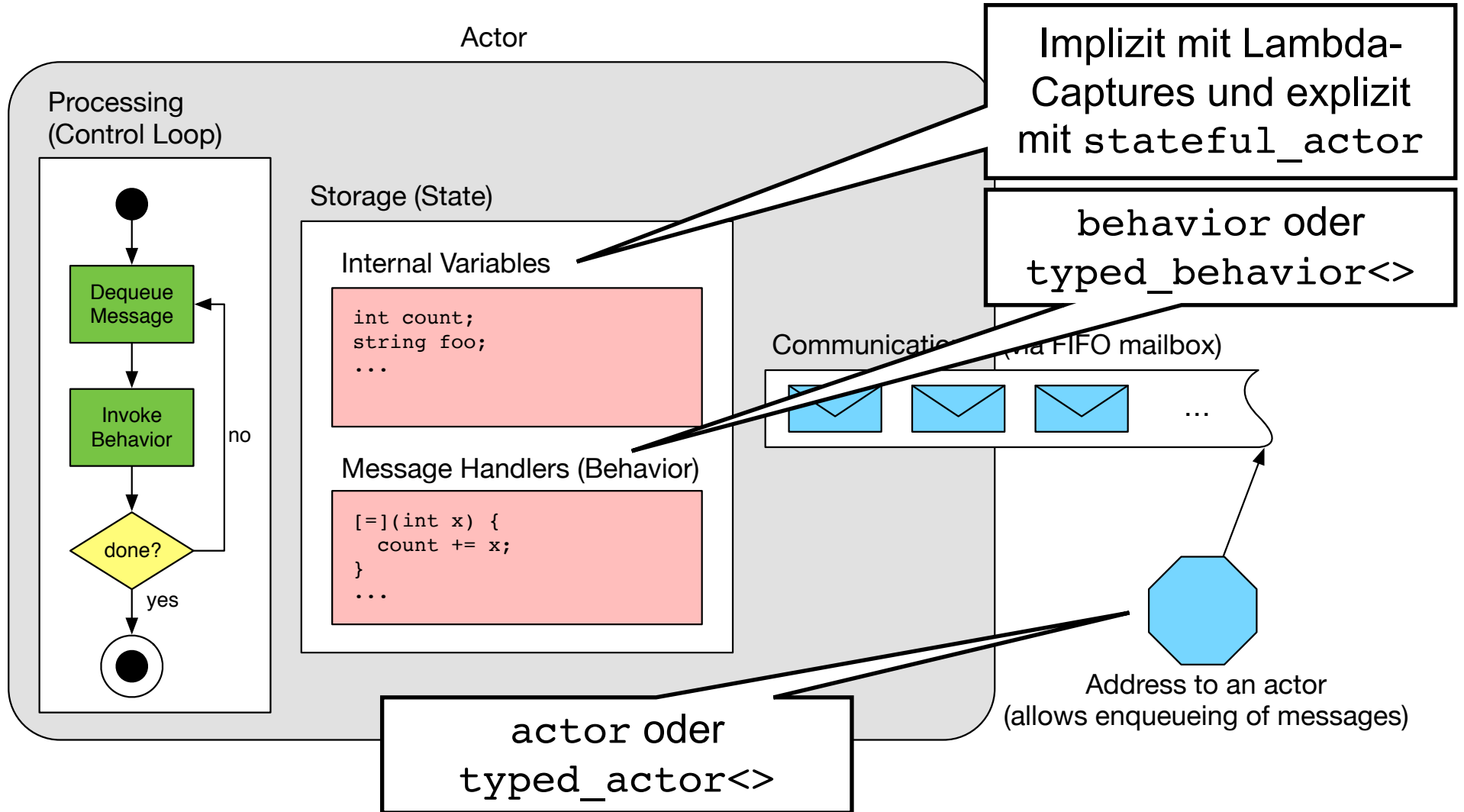
void caf_main(actor_system&, const config& c);
```

Eigene Konfigurations-  
Klasse mit Parametern  
als Member-Variablen

Legt neue Parameter-  
Gruppe „global“; fügt  
msg unter dem Namen  
„message“ hinzu mit  
-m als Kurzform für  
Kommandozeilen

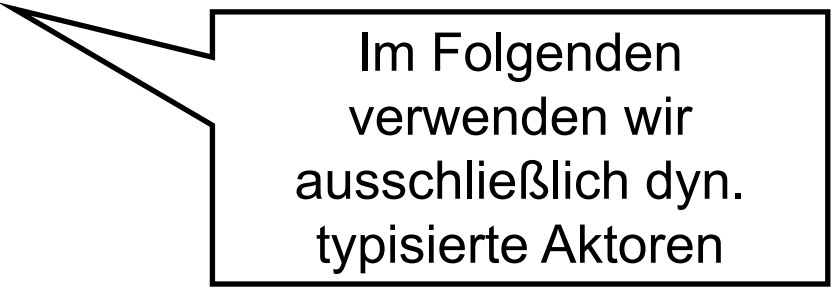
caf\_main erlaub  
benutzerdefinierte  
Konfiguration als  
zweites Argument

# Anatomie eines CAF Aktors



# Arten von Aktoren in CAF

- ◆ CAF unterscheidet dynamisch und statisch typisierte Aktoren
- ◆ Dynamisch typisierte Aktoren akzeptieren alle Nachrichten
  - `actor` identifiziert dynamisch typisierte Aktoren
  - Keine Typprüfung beim Sender, potentiell Laufzeitfehler
- ◆ Statisch typisierte Aktoren definieren ein *Messaging Interface*
  - `typed_actor<>` identifiziert statisch typisierte Aktoren
  - Compiler prüft Ein- und Ausgabenachrichten



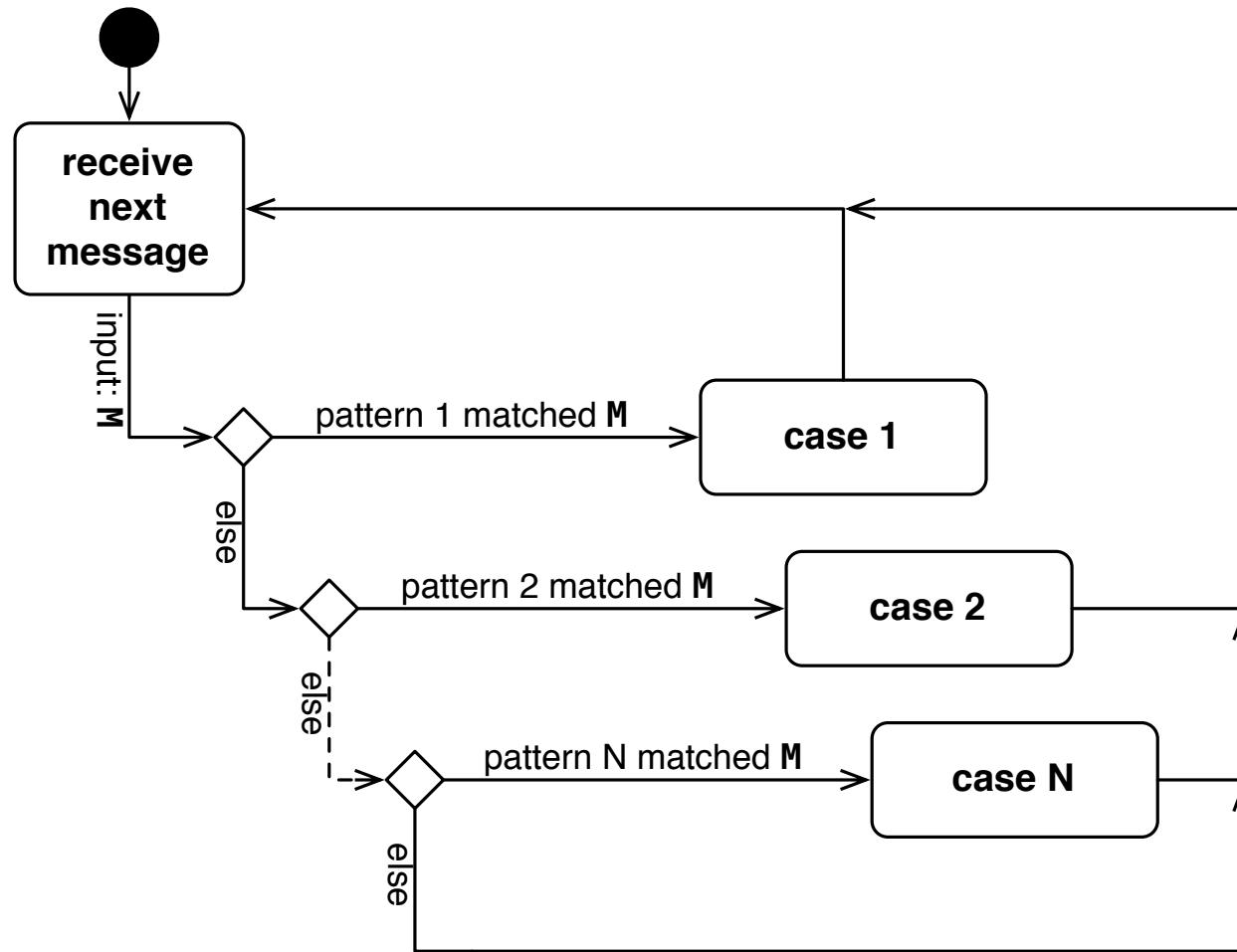
Im Folgenden  
verwenden wir  
ausschließlich dyn.  
typisierte Aktoren



# Event-basierte Aktoren

- ◆ Kein expliziter Empfang von Nachrichten
  - Kontrollfluss ist der Laufzeit überlassen
  - Aktoren arbeiten ihre Mailbox ab bis sie terminieren
- ◆ Aktoren definieren ein `behavior` mit Nachrichten-Handlern
  - Signatur der Nachrichtenhandler fungiert als *Pattern*
  - Rückgabewert der Handler generiert Antwortnachrichten

# Event-basierter Kontrollfluss



# Simpler Additions-Aktor

```
behavior
adder(event_based_actor* self) {
    return {
        [](int x, int y) {
            return x + y;
        }
    };
}
```

Ein Aktor in CAF ist typischerweise implementiert als Funktion, die behavior zurück gibt und optional einen self-Pointer als erstes Argument nimmt

Rückgabewert des Nachrichten-Handlers ist die Antwortnachricht

# Kommunikation in CAF

- ◆ Asynchroner Nachrichtenversand: `send`
  - Eventuelle Antworten im `behavior` verarbeitet
  - Syntax: `self->send(other, ...);`
- ◆ Request/Response: `request`
  - Antwort mit dediziertem „one-shot Handler“ verarbeitet
  - Syntax: `self->request(other, ...).then(...);`
- ◆ Weiterreichen erhaltener Aufgaben: `delegate`
  - Transparentes Forwarding
  - Syntax: `self->delegate(other, ...);`

# Ad hoc Kommunikation mit Aktoren

- ◆ `scoped_actor` erlaubt Komm. mit Aktoren von „außerhalb“
- ◆ Ad hoc Aktor um z.B. aus `main` heraus zu kommunizieren
- ◆ Expliziter, blockierender Empfang von Nachrichten
- ◆ Erfordert ein `actor_system` zur Erschaffung:

```
void caf_main(actor_system& sys) {  
    scoped_actor self{sys};  
}
```

- ◆ Hinweis: blockierende Aktoren kennen kein `.then`:  
`self->request(other, ...).receive(...);`

# Starten von und reden mit Aktoren

```
actor a = sys.spawn(adder);  
scoped_actor self{sys};  
self->request(a, infinite, 1, 2)  
  .receive(  
    [&](int z) {  
      cout << "1+2=" << z << "\n";  
    },  
    [&](error& err) {  
      cout << "Error: "  
        << to_string(err)  
        << "\n"; });
```

Startet eine Instanz des  
simplen Additions-  
Aktors

Schickt Tupel (1, 2) als  
Request an a über den  
ad hoc Aktor self ohne  
timeout (infinite);  
anschließend warten  
auf Antwort mit  
.receive

Handler wird aufgerufen  
bei Fehlern im  
Empfänger oder wenn  
ein Timeout auftritt

# Aktor für Grundrechenarten

```
behavior math() {  
    return {  
        [](int x, int y) {  
            return x + y;  
        },  
        [](int x, int y) {  
            return x - y;  
        },  
        [](int x, int y) {  
            return x * y;  
        },  
    };  
}
```

Wir schicken dem Aktor die  
Nachricht (3, 23). Wo  
landet diese?

Problem:  
Nachrichten landen  
immer hier!

# Diskussion des Math-Aktors

- ◆ Zwei `int` als Eingabe geben keinen Hinweis auf die Semantik
- ◆ Erweitern mit neuen Operationen nicht möglich (z.B. Division)
- ◆ Idee: Annotieren/Typisieren der Daten
  - Möglichkeit #1: Eine Klasse pro Operation
    - + kein Laufzeit-Overhead
    - viel Boilerplate-Code erforderlich
  - Möglichkeit #2: Eineindeutig typisierte Metadaten
    - + wenig/kein Laufzeit-Overhead
    - + wenig Boilerplate-Code erforderlich



# Atoms

- ◆ Erlauben das leichtgewichtige Annotieren von Daten
- ◆ Ermöglichen Definition von Compilezeit Konstanten
- ◆ Mit `CAF_ADD_ATOM` eineindeutig typisierte Konstanten:  
`CAF_ADD_ATOM(vslab, add_atom)`
- ◆ Dies muss innerhalb des Type ID Blocks passieren
- ◆ Instanz: `add_atom_v`



# Metadaten für Math. Operationen

- ◆ Type sind nummeriert und sollten nicht überlappen!
- ◆ CAF hat eine Konstante für die erste freie ID
- ◆ Eineindeutig typisierte Konstanten für alle Grundrechenarten:

```
CAF_BEGIN_TYPE_ID_BLOCK(vslab,  
                        first_custom_type_id)  
CAF_ADD_ATOM(vslab, vs, add_atom)  
CAF_ADD_ATOM(vslab, vs, sub_atom)  
CAF_ADD_ATOM(vslab, vs, mul_atom)  
CAF_ADD_ATOM(vslab, vs, div_atom)  
CAF_END_TYPE_ID_BLOCK(vslab)
```

Atomname

Namespace

Projektname

# Aktor für Grundrechenarten

```
behavior math() {  
    return {  
        [](add_atom, int x, int y) {  
            return x + y;  
        },  
        [](sub_atom, int x, int y) {  
            return x - y;  
        },  
        [](mul_atom, int x, int y) {  
            return x * y;  
        },  
    }  
}
```

Konvention: Metadaten,  
vor Daten

## Aktor für Grundrechenarten (Forts.)

```
[ ](div_atom, int x, int y)
-> result<int> {
    if (y == 0)
        return sec::invalid_argument;
    return x / y;
}
};
}
```

Repräsentiert einen `int`  
oder einen Fehler

SEC steht für „System  
Error Code“

# Verteilte Aktoren

- ◆ Lernen verteilter Aktoren durch Rendezvous-Prozess:
  - „Server“ wird mit `publish` an einen Port gebunden
  - „Client“ lernt Server-Handle mit `remote_actor`
  - Client/Server-Rollen nur für Rendezvous erforderlich
- ◆ Akteur-Handles können im Netzwerk versendet werden
- ◆ C++ hat *keine* Reflections: Benutzerdefinierte Datentypen müssen CAF explizit bekannt gemacht werden
  - Dies beinhaltet insb. auch Instanziierungen für `vector<T>`
  - Datentypen werden per Konfiguration hinzugefügt

# Konfiguration verteilter Aktoren

```
struct config : actor_system_config {
    std::string host = "localhost";
    uint16_t port = 0;
    bool server = false;
    config() {
        opt_group{custom_options_, "global"}
            .add(host, "host,H", "hostname of server")
            .add(port, "port,p", "IP port")
            .add(server, "server,s", "run as server");
    }
};
```

# Entfernter math-Aktor

```
auto& mm = sys.middleman();  
if (cfg.server) {  
    auto a = sys.spawn(math);  
    auto p = mm.publish(a,  
                        cfg.port);  
    ...  
} else {  
    auto x = mm.remote_actor(cfg.host,  
                             cfg.port);  
    if (x) {  
        ...  
    }  
}
```

Referenz auf die  
Netzwerk-Komponente

Macht a ansprechbar  
über konfigurierten Port;  
gibt gebundenen Port  
zurück bei Erfolg

Gibt ein expected  
zurück: bei Erfolg ein  
Handle, sonst Fehler

# Eigene Datentypen (Compilezeit)

```
struct point2d {  
    int x;  
    int y;  
};
```

```
template <class Inspector>  
bool inspect(Inspector& f, point2d& p) {  
    return f.object(p).fields(f.field("x", p.x),  
                             f.field("y", p.y));  
}
```

inspect erlaubt CAF  
Serialisierung und  
String-Konvertierung

Der Inspector wird  
aufgerufen mit allen  
Datenfeldern und  
Metadaten



# Eigene Datentypen (Compilezeit)

- ◆ Werden im selben Type ID Block bekannt gemacht:

```
CAF_BEGIN_TYPE_ID_BLOCK(vslab,  
                          first_custom_type_id)  
CAF_ADD_TYPE_ID(vslab, (point2d))  
CAF_END_TYPE_ID_BLOCK(vslab)
```

# Expliziter State in Aktoren

- ◆ CAF erlaubt klare Trennung von State und Verhalten
- ◆ Modellierung im Code ist explizit:
  - Datenfelder in separater Klasse `T` zusammengefasst
  - Der `self`-Pointer ist vom Typ `stateful_actor<T>`
  - Zugriff auf den State geschieht mit `self->state`
- ◆ State wird erzeugt beim starten und zerstört beim beenden
  - Wichtiger Unterschied: der Akteur selbst wird erst zerstört sobald keine Referenzen auf ihn mehr vorhanden sind, der State wird zerstört sobald `self->quit` aufgerufen wird

# Aktor mit State: Datenzelle

```
struct cell_state {  
    int value = 0;  
};  
behavior cell(stateful_actor<cell_state>* self) {  
    return {  
        [=](get_atom) {  
            return self->state.value;  
        },  
        [=](put_atom, int x) {  
            self->state.value = x;  
        }  
    };  
}
```

CAF startet Aktoren  
abhängig von der  
Funktionssignatur

Kopiert den self-  
Pointer in die anonyme  
Funktion

# Monitoring

- ◆ Aktoren werden überwacht mit `self->monitor(x)`
- ◆ Wenn `x` beendet wird: Zustellung einer `down_msg`
- ◆ Mehrfaches Monitoring führt zu multiplen `down_msg`
- ◆ Kurznotation um Aktoren zu starten und zu überwachen:  
`self->spawn<monitored>( ... )`
- ◆ Unidirektional: keine Signalisierung an überwachte Aktoren
- ◆ Eine `down_msg` wird in separatem Handler verarbeitet:  
`self->set_down_handler( ... );`

# Linking

- ◆ Aktoren werden verlinkt mit `self->link_to(x)`
- ◆ Wenn x beendet wird: Zustellung einer `exit_msg`
- ◆ Mehrfaches Linken hat keinen Effekt
- ◆ Kurznotation um Aktoren für starten und überwachen:  
`self->spawn<linked>( ... )`
- ◆ Bidirektional: beide Aktoren erhalten jeweils `exit_msg`
- ◆ Eine `exit_msg` wird in separatem Handler verarbeitet:  
`self->set_exit_handler( ... );`

# Gruppenkommunikation

- ◆ Zur losen Kopplung von Aktoren unterstützt CAF Publish/Subscribe-basierte Gruppenkommunikation
- ◆ Aktoren treten Gruppen mit `self->join(grp)` bei:
  - Empfang aller Nachrichten im regulären `behavior`
  - Nachricht an **alle** Teilnehmer: `self->send(grp, ...)`
- ◆ Verteilung ist analog zu `publish` und `remote_actor`
  - Ein Server ruft `publish_local_groups` auf
  - Clients verbinden sich mit `remote_group`

# Wichtige Online-Quellen zu CAF

- ◆ HTML Handbuch: <http://actor-framework.readthedocs.io>
  - Nach Komponente/Thema sortiertes Referenzbuch
  - Beinhaltet Anforderungen, Installationsguide, etc.
- ◆ Repository: <https://github.com/actor-framework/actor-framework>
  - Quellcode und CMake-Setup für die Hauptkomponenten
  - Beispielprogramme (im Verzeichnis *examples*)
- ◆ Doxygen: <https://codedocs.xyz/actor-framework/actor-framework/>