libcppa

A C++ library for actor programming

User Manual

`libcppa` version 0.8.2

Dominik Charousset

March 18, 2014

Contents

1 First Steps

To compile *libcppa*, you will need CMake and a C++11 compiler. To get and compile the sources, open a terminal (on Linux or Mac OS X) and type:

```
git clone git://github.com/Neverlord/libcppa.git
cd libcppa
./configure
make
make install [as root, optional]
```

It is recommended to run the unit tests as well:

```
make test
```

Please submit a bug report that includes (a) your compiler version, (b) your OS, and (c) the content of the file `build/Testing/Temporary/LastTest.log` if an error occurs.

1.1 Features Overview

- Lightweight, fast and efficient actor implementations
- Network transparent messaging
- Error handling based on Erlang's failure model
- Pattern matching for messages as internal DSL to ease development
- Thread-mapped actors and on-the-fly conversions for soft migration of existing applications
- Publish/subscribe group communication

1.2 Supported Compilers

- GCC $\geq$ 4.7
- Clang $\geq$ 3.2

1.3 Supported Operating Systems

- Linux
- Mac OS X
- *Note for MS Windows*: *libcppa* relies on C++11 features such as variadic templates. We will support this platform as soon as Microsoft's compiler implements all required C++11 features.

1.4 Hello World Example

```
#include <string>
#include <iostream>
#include "cppa/cppa.hpp"

using namespace std;
using namespace cppa;

void mirror() {
  // wait for messages
  become (
    // invoke this lambda expression if we receive a string
    on_arg_match >> [](const string& what) -> string {
      // prints "Hello World!" via aout (thread-safe cout wrapper)
      aout << what << endl;
      // terminates this actor afterwards;
      // 'become' otherwise loops forever
      self->quit();
      // replies "!dlroW olleH"
      return string(what.rbegin(), what.rend());
    }
  );
}

void hello_world(const actor_ptr& buddy) {
  // send "Hello World!" to our buddy ...
  sync_send(buddy, "Hello World!").then(
    // ... and wait for a response
    on_arg_match >> [](const string& what) {
      // prints "!dlroW olleH"
      aout << what << endl;
    }
  );
}

int main() {
  // create a new actor that calls 'mirror()'
  auto mirror_actor = spawn(mirror);
  // create another actor that calls 'hello_world(mirror_actor)'
  spawn(hello_world, mirror_actor);
  // wait until all other actors we have spawned are done
  await_all_others_done();
  // run cleanup code before exiting main
  shutdown();
}
```

2 Copy-On-Write Tuples

The message passing implementation of *libcppa* uses tuples with call-by-value semantic. Hence, it is not necessary to declare message types, though, *libcppa* allows users to use user-defined types in messages (see Section 13.1). A call-by-value semantic would cause multiple copies of a tuple if it is send to multiple actors. To avoid unnecessary copying overhead, *libcppa* uses a copy-on-write tuple implementation. A tuple is implicitly shared between any number of actors, as long as all actors demand only read access. Whenever an actor demands write access, it has to copy the data first if more than one reference to it exists. Thus, race conditions cannot occur and each tuple is copied only if necessary.

The interface of `cow_tuple` strictly distinguishes between const and non-const access. The template function `get` returns an element as immutable value, while `get_ref` explicitly returns a mutable reference to the required value and detaches the tuple if needed. We do not provide a const overload for `get`, because this would cause to unintended, and thus unnecessary, copying overhead.

```
auto x1 = make_cow_tuple(1, 2, 3);      // cow_tuple<int, int, int>
auto x2 = x1;                           // cow_tuple<int, int, int>
assert(&get<0>(x1) == &get<0>(x2));     // point to the same data
get_ref<0>(x1) = 10;                    // detaches x1 from x2
//get<0>(x1) = 10;                      // compiler error
assert(get<0>(x1) == 10);               // x1 is now {10, 2, 3}
assert(get<0>(x2) == 1);                // x2 is still {1, 2, 3}
assert(&get<0>(x1) != &get<0>(x2));     // no longer the same
```

2.1 Dynamically Typed Tuples

The class `any_tuple` represents a tuple without static type information. All messages send between actors use this tuple type. The type information can be either explicitly accessed for each element or the original tuple, or a subtuple of it, can be restored using `tuple_cast`. Users of *libcppa* usually do not need to know about `any_tuple`, since it is used “behind the scenes”. However, `any_tuple` can be created from a `cow_tuple` or by using `make_any_tuple`, as shown below.

```
auto x1 = make_cow_tuple(1, 2, 3);      // cow_tuple<int, int, int>
any_tuple x2 = x1;                      // any_tuple
any_tuple x3 = make_cow_tuple(10, 20); // any_tuple
auto x4 = make_any_tuple(42);           // any_tuple
```

2.2 Casting Tuples

The function `tuple_cast` restores static type information from an `any_tuple` object. It returns an `option` (see Section 17.1) for a `cow_tuple` of the requested types.

```
auto x1 = make_any_tuple(1, 2, 3);
auto x2_opt = tuple_cast<int, int, int>(x1);
assert(x2_opt.valid());
auto x2 = *x2_opt;
assert(get<0>(x2) == 1);
assert(get<1>(x2) == 2);
assert(get<2>(x2) == 3);
```

The function `tuple_cast` can be used with wildcards (see Section 3.4) to create a view to a subset of the original data. No elements are copied, unless the tuple becomes detached.

```
auto x1 = make_cow_tuple(1, 2, 3);
any_tuple x2 = x1;
auto x3_opt = tuple_cast<int, anything, int>(x2);
assert(x3_opt.valid());
auto x3 = *x3_opt;
assert(get<0>(x3) == 1);
assert(get<1>(x3) == 3);
assert(&get<0>(x3) == &get<0>(x1));
assert(&get<1>(x3) == &get<2>(x1));
```

3 Pattern Matching

C++ does not provide pattern matching facilities. A general pattern matching solution for arbitrary data structures would require a language extension. Hence, we decided to restrict our implementation to tuples, to be able to use an internal domain-specific language approach.

3.1 Basics

A match expression begins with a call to the function `on`, which returns an intermediate object providing the member function `when` and `operator>>`. The right-hand side of the operator denotes a callback, usually a lambda expression, that should be invoked if a tuple matches the types given to `on`, as shown in the example below.

```
on<int>() >> [](int i) { /*...*/ }
on<int, float>() >> [](int i, float f) { /*...*/ }
on<int, int, int>() >> [](int a, int b, int c) { /*...*/ }
```

The result of `operator>>` is a *match statement*. A partial function can consist of any number of match statements. At most one callback is invoked, since the evaluation stops at the first match.

```
partial_function fun {
  on<int>() >> [](int i) {
    // case1
  },
  on<int>() >> [](int i) {
    // case2; never invoked, since case1 always matches first
  }
};
```

The function “`on`” can be used in two ways. Either with template parameters only or with function parameters only. The latter version deduces all types from its arguments and matches for both type and value. To match for any value of a given type, “`val`” can be used, as shown in the following example.

```
on(42) >> [](int i) { assert(i == 42); }
on("hello world") >> [] { /* ... */ }
on("print", val<std::string>) >> [](const std::string& what) {
  // ...
}
```

Note: The given callback can have less arguments than the pattern. But it is only allowed to skip arguments from left to right.

```
on<int, float, double>() >> [](double) { /*...*/ }              // ok
on<int, float, double>() >> [](float, double) { /*...*/ }       // ok
on<int, float, double>() >> [](int, float, double) { /*...*/ } // ok

on<int, float, double>() >> [](int i) { /*...*/ } // compiler error
```

3.2 Reducing Redundancy with "`arg_match`" and "`on_arg_match`"

Our previous examples always used the most verbose form, which is quite redundant, since you have to type the types twice – as template parameter and as argument type for the lambda. To avoid such redundancy, `arg_match` can be used as last argument to the function `on`. This causes the compiler to deduce all further types from the signature of the given callback.

```
on<int, int>() >> [](int a, int b) { /*...*/ }
// is equal to:
on(arg_match) >> [](int a, int b) { /*...*/ }
```

Note that the second version does call `on` without template parameters. Furthermore, `arg_match` must be passed as last parameter. If all types should be deduced from the callback signature, `on_arg_match` can be used. It is equal to `on(arg_match)`.

```
on_arg_match >> [](const std::string& str) { /*...*/ }
```

3.3 Atoms

Assume an actor provides a mathematical service for integers. It takes two arguments, performs a predefined operation and returns the result. It cannot determine an operation, such as multiply or add, by receiving two operands. Thus, the operation must be encoded into the message. The Erlang programming language introduced an approach to use non-numerical constants, so-called *atoms*, which have an unambiguous, special-purpose type and do not have the runtime overhead of string constants. Atoms are mapped to integer values at compile time in *libcppa*. This mapping is guaranteed to be collision-free and invertible, but limits atom literals to ten characters and prohibits special characters. Legal characters are "`_0-9A-Za-z`" and the whitespace character. Atoms are created using the `constexpr` function `atom`, as the following example illustrates.

```
on(atom("add"), arg_match) >> [](int a, int b) { /*...*/ },
on(atom("multiply"), arg_match) >> [](int a, int b) { /*...*/ },
// ...
```

Note: The compiler cannot enforce the restrictions at compile time, except for a length check. The assertion `atom("!?") != atom("?!")` is not true, because each invalid character is mapped to the whitespace character.

3.4 Wildcards

The type `anything` can be used as wildcard to match any number of any types. A pattern created by `on<anything>()` or its alias `others()` is useful to define a default case. For patterns defined without template parameters, the `constexpr` value `any_vals` can be used as function argument. The constant `any_vals` is of type `anything` and is nothing but syntactic sugar for defining patterns.

```
on<int, anything>() >> [](int i) {
  // tuple with int as first element
},
on(any_vals, arg_match) >> [](int i) {
  // tuple with int as last element
  // "on(any_vals, arg_match)" is equal to "on(anything{}, arg_match)"
},
others() >> [] {
  // everything else (default handler)
  // "others()" is equal to "on<anything>()" and "on(any_vals)"
}
```

3.5 Guards

Guards can be used to constrain a given match statement by using placeholders, as the following example illustrates.

```
using namespace cppa::placeholders; // contains _x1 - _x9

on<int>().when(_x1 % 2 == 0) >> [] {
  // int is even
},
on<int>() >> [] {
  // int is odd
}
```

Guard expressions are a lazy evaluation technique. The placeholder `_x1` is substituted with the first value of a given tuple. All binary comparison and arithmetic operators are supported, as well as `&&` and `||`. In addition, there are three functions designed to be used in guard expressions: `gref` (“guard reference”), `gval` (“guard value”), and `gcall` (“guard function call”). The function `gref` creates a reference wrapper, while `gval` encloses a value. It is similar to `std::ref` but it is always `const` and “lazy”. A few examples to illustrate some pitfalls:

```
int val = 42;

on<int>().when(_x1 == val)           // (1) matches if _x1 == 42
on<int>().when(_x1 == gref(val))     // (2) matches if _x1 == val
on<int>().when(_x1 == std::ref(val)) // (3) ok, because of placeholder
others().when(gref(val) == 42)       // (4) matches everything
                                     //     as long as val == 42
others().when(std::ref(val) == 42)   // (5) compiler error
```

Statement (5) is evaluated immediately and returns a boolean, whereas statement (4) creates a valid guard expression. Thus, you should always use gref instead of std::ref to avoid errors.

The second function, gcall, encapsulates a function call. Its usage is similar to std::bind, but there is also a short version for unary functions: gcall(fun, _x1) is equal to _x1(fun).

```
auto vec_sorted = [](const std::vector<int>& vec) {
  return std::is_sorted(vec.begin(), vec.end());
};

on<std::vector<int>>().when(gcall(vec_sorted, _x1)) // is equal to:
on<std::vector<int>>().when(_x1(vec_sorted)))
```

3.5.1 Placeholder Interface

```
template<int X>
struct guard_placeholder;
```

Member functions (x represents the value at runtime, y represents an iterable container)

size()	Returns x.size()
empty()	Returns x.empty()
not_empty()	Returns !x.empty()
front()	Returns an option (see Section 17.1) to x.front()
in(y)	Returns true if y contains x, false otherwise
not_in(y)	Returns !in(y)

3.5.2 Examples for Guard Expressions

```
using namespace std;
typedef vector<int> ivec;

vector<string> strings{"abc", "def"};

on_arg_match.when(_x1.front() == 0) >> [](const ivec& v) {
  // note: we don't have to check whether _x1 is empty in our guard,
  //       because '_x1.front()' returns an option for a
  //       reference to the first element
  assert(v.size() >= 1);
  assert(v.front() == 0);
},
on<int>().when(_x1.in({10, 20, 30})) >> [](int i) {
  assert(i == 10 || i == 20 || i == 30);
},
on<string>().when(_x1.not_in(strings)) >> [](const string& str) {
  assert(str != "abc" && str != "def");
},
on<string>().when(_x1.size() == 10) >> [](const string& str) {
  // ...
}
```

3.6 Projections and Extractors

Projections perform type conversions or extract data from a given input. If a callback expects an integer but the received message contains a string, a projection can be used to perform a type conversion on-the-fly. This conversion should be free of side-effects and, in particular, shall not throw exceptions, because a failed projection is not an error. A pattern simply does not match if a projection failed. Let us have a look at a simple example.

```
auto intproj = [](const string& str) -> option<int> {
  char* endptr = nullptr;
  int result = static_cast<int>(strtol(str.c_str(), &endptr, 10));
  if (endptr != nullptr && *endptr == '\0') return result;
  return {};
};
partial_function fun {
  on(intproj) >> [](int i) {
    // case 1: successfully converted a string
  },
  on_arg_match >> [](const string& str) {
    // case 2: str is not an integer
  }
};
```

The lambda `intproj` is a `string` ⇒ `int` projection, but note that it does not return an integer. It returns `option<int>`, because the projection is not guaranteed to always succeed. An empty `option` indicates, that a value does not have a valid mapping to an integer. A pattern does not match if a projection failed.

Note: Functors used as projection must take exactly one argument and must return a value. The types for the pattern are deduced from the functor's signature. If the functor returns an `option<T>`, then `T` is deduced.

4 Actors

libcppa provides several actor implementations, each covering a particular use case. The class `local_actor` is the base class for all implementations, except for (remote) proxy actors. Hence, `local_actor` provides a common interface for actor operations like trapping exit messages or finishing execution. The default actor implementation in *libcppa* is event-based. Event-based actors have a very small memory footprint and are thus very lightweight and scalable. Context-switching actors are used for actors that make use of the blocking API (see Section 14), but do not need to run in a separate thread. Context-switching and event-based actors are scheduled cooperatively in a thread pool. Thread-mapped actors can be used to opt-out of this cooperative scheduling.

4.1 The "Keyword" `self`

The `self` pointer is an essential ingredient of our design. It identifies the running actor similar to the implicit `this` pointer identifying an object within a member function. Unlike `this`, though, `self` is not limited to a particular scope. The `self` pointer is used implicitly, whenever an actor calls functions like `send` or `become`, but can be accessed to use more advanced actor operations such as linking to another actor, e.g., by calling `self->link_to(other)`. The `self` pointer is convertible to `actor_ptr` and `local_actor*`, but it is neither copyable nor assignable. Thus, `auto s = self` will cause a compiler error, while `actor_ptr s = self` works as expected.

A thread that accesses `self` is converted on-the-fly to an actor if needed. Hence, "everything is an actor" in *libcppa*. However, automatically converted actors use an implementation based on the blocking API, which behaves slightly different than the default, i.e., event-based, implementation.

4.2 Interface

```
class local_actor;
```

Member functions

`quit(uint32_t reason = normal)`	Finishes execution of this actor

Observers

`bool trap_exit()`	Checks whether this actor traps exit messages
`bool chaining()`	Checks whether this actor uses the “chained send” optimization (see Section 5.2)
`any_tuple last_dequeued()`	Returns the last message that was dequeued from the actor’s mailbox **Note**: Only set during callback invocation
`actor_ptr last_sender()`	Returns the sender of the last dequeued message **Note**$_1$: Only set during callback invocation **Note**$_2$: Used implicitly to send response messages (see Section 5.1)
`vector<group_ptr> joined_groups()`	Returns all subscribed groups

Modifiers

`void trap_exit(bool enabled)`	Enables or disables trapping of exit messages
`void chaining(bool enabled)`	Enables or disables chained send
`void join(const group_ptr& g)`	Subscribes to group `g`
`void leave(const group_ptr& g)`	Unsubscribes group `g`
`auto make_response_handle()`	Creates a handle that can be used to respond to the last received message later on, e.g., after receiving another message
`void on_sync_failure(auto fun)`	Sets a handler, i.e., a functor taking no arguments, for unexpected synchronous response messages (default action is to kill the actor for reason `unhandled_sync_failure`)
`void on_sync_timeout(auto fun)`	Sets a handler, i.e., a functor taking no arguments, for `timed_sync_send` timeout messages (default action is to kill the actor for reason `unhandled_sync_timeout`)
`void monitor(actor_ptr whom)`	Adds a unidirectional monitor to `whom` (see Section 8.2)
`void demonitor(actor_ptr whom)`	Removes a monitor from `whom`
`void exec_behavior_stack()`	Executes an actor’s behavior stack until it is empty
`bool has_sync_failure_handler()`	Checks wheter this actor has a user-defined sync failure handler

5 Sending Messages

Messages can be sent by using send, send_tuple, or operator<<. The variadic template function send has the following signature.

```
template<typename... Args>
void send(actor_ptr whom, Args&&... what);
```

The variadic template pack what... is converted to a dynamically typed tuple (see Section 2.1) and then enqueued to the mailbox of whom. The following example shows two equal sends, one using send and the other using operator<<.

```
actor_ptr other = spawn(...);
send(other, 1, 2, 3);
other << make_any_tuple(1, 2, 3);
```

Using the function send is more compact, but does not have any other benefit. However, note that you should not use send if you already have an instance of any_tuple, because it creates a new tuple containing the old one.

```
actor_ptr other = spawn(...);
auto msg = make_any_tuple(1, 2, 3);
send(other, msg); // oops, creates a new tuple that contains msg
send_tuple(other, msg); // ok
other << msg; // ok
```

The function send_tuple is equal to operator<<. Choosing one or the other is merely a matter of personal preferences.

5.1 Replying to Messages

The return value of a message handler is used as response message. During callback invokation, `self->last_sender()` is set. This identifies the sender of the received message and is used implicitly to reply to the correct sender. However, using `send(self->last_sender(), ...)` does *not* reply to the message, i.e., synchronous messages will not recognize the message as response.

```
void client(const actor_ptr& master) {
  become (
    on("foo", arg_match) >> [=](const string& request) -> string {
      return sync_send(master, atom("bar"), request).then(
        on_arg_match >> [=](const std::string& response) {
          return response;
        }
      );
    }
  );
};
```

5.2 Chaining

Sending a message to a cooperatively scheduled actor usually causes the receiving actor to be put into the scheduler's job queue if it is currently blocked, i.e., is waiting for a new message. This job queue is accessed by worker threads. The *chaining* optimization does not cause the receiver to be put into the scheduler's job queue if it is currently blocked. The receiver is stored as successor of the currently running actor instead. Hence, the active worker thread does not need to access the job queue, which significantly speeds up execution. However, this optimization can be inefficient if an actor first sends a message and then starts computation.

```
void foo(actor_ptr other) {
  send(other, ...);
  very_long_computation();
  // ...
}

int main() {
  // ...
  auto a = spawn(...);
  auto b = spawn(foo, a);
  // ...
}
```

The example above illustrates an inefficient work flow. The actor `other` is marked as successor of the `foo` actor but its execution is delayed until `very_long_computation()` is done. In general, actors should follow the work flow `receive` ⇒`compute` ⇒ `send results`. However, this optimization can be disabled by calling `self->chaining(false)` if an actor does not match this work flow.

```
void foo(actor_ptr other) {
  self->chaining(false);    // disable chaining optimization
  send(other, ...);         // not delayed by very_long_compuation
  very_long_computation();
  // ...
}
```

5.3 Delaying Messages

Messages can be delayed, e.g., to implement time-based polling strategies, by using one of `delayed_send`, `delayed_send_tuple`, `delayed_reply`, or `delayed_reply_tuple`. The following example illustrates a polling strategy using `delayed_send`.

```
delayed_send(self, std::chrono::seconds(1), atom("poll"));
become (
  on(atom("poll")) >> []() {
    // poll a resource...
    // schedule next polling
    delayed_send(self, std::chrono::seconds(1), atom("poll"));
  }
);
```

5.4 Forwarding Messages

The function `forward_to` forwards the last dequeued message to an other actor. Forwarding a synchronous message will also transfer responsibility for the request, i.e., the receiver of the forwarded message can reply as usual and the original sender of the message will receive the response. The following diagram illustrates forwarding of a synchronous message from actor `B` to actor `C`.

```
A                  B                  C
|                  |                  |
| --(sync_send)--> |                  |
|                  | --(forward_to)-> |
|                  X                  |---\
|                                     |   | compute
|                                     |   | result
|                                     |<--/
| <-------------(reply)-------------- |
|                                     X
|---\
|   | handle
|   | response
|<--/
|
X
```

The forwarding is completely transparent to actor `C`, since it will see actor `A` as sender of the message. However, actor `A` will see actor `C` as sender of the response message instead of actor `B` and thus could recognize the forwarding by evaluating `self->last_sender()`.

6 Receiving Messages

The current *behavior* of an actor is its response to the *next* incoming message and includes (a) sending messages to other actors, (b) creation of more actors, and (c) setting a new behavior.

An event-based actor, i.e., the default implementation in *libcppa*, uses `become` to set its behavior. The given behavior is then executed until it is replaced by another call to `become` or the actor finishes execution.

6.1 Class-based actors

A class-based actor is a subtype of `event_based_actor` and must implement the pure virtual member function `init`. An implementation of `init` shall set an initial behavior by using `become`.

```
class printer : public event_based_actor {
  void init() {
    become (
      others() >> [] {
        cout << to_string(self->last_received()) << endl;
      }
    );
  }
};
```

Another way to implement class-based actors is provided by the class `sb_actor` ("State-Based Actor"). This base class calls `become(init_state)` in its `init` member function. Hence, a subclass must only provide a member of type `behavior` named `init_state`.

```
struct printer : sb_actor<printer> {
  behavior init_state = (
    others() >> [] {
      cout << to_string(self->last_received()) << endl;
    }
  );
};
```

Note that `sb_actor` uses the Curiously Recurring Template Pattern. Thus, the derived class must be given as template parameter. This technique allows `sb_actor` to access the `init_state` member of a derived class. The following example illustrates a more advanced state-based actor that implements a stack with a fixed maximum number of elements.

```
class fixed_stack : public sb_actor<fixed_stack> {

    friend class sb_actor<fixed_stack>;

    size_t max_size = 10;

    vector<int> data;

    behavior full;
    behavior filled;
    behavior empty;

    behavior& init_state = empty;

 public:

    fixed_stack(size_t max) : max_size(max)  {
        full = (
            on(atom("push"), arg_match) >> [=](int) { /* discard */ },
            on(atom("pop")) >> [=]() -> cow_tuple<atom_value, int> {
                auto result = data.back();
                data.pop_back();
                become(filled);
                return {atom("ok"), result};
            }
        );
        filled = (
            on(atom("push"), arg_match) >> [=](int what) {
                data.push_back(what);
                if (data.size() == max_size) become(full);
            },
            on(atom("pop")) >> [=]() -> cow_tuple<atom_value, int> {
                auto result = data.back();
                data.pop_back();
                if (data.empty()) become(empty);
                return {atom("ok"), result};
            }
        );
        empty = (
            on(atom("push"), arg_match) >> [=](int what) {
                data.push_back(what);
                become(filled);
            },
            on(atom("pop")) >> [=] {
                return atom("failure");
            }
        );

    }

};
```

6.2 Nesting Receives Using `become/unbecome`

Since `become` does not block, an actor has to manipulate its behavior stack to achieve nested receive operations. An actor can set a new behavior by calling `become` with the `keep_behavior` policy to be able to return to its previous behavior later on by calling `unbecome`, as shown in the example below.

```
// receives {int, float} sequences
void testee() {
  become (
    on_arg_match >> [=](int value1) {
      become (
        // the keep_behavior policy stores the current behavior
        // on the behavior stack to be able to return to this
        // behavior later on by calling unbecome()
        keep_behavior,
        on_arg_match >> [=](float value2) {
          cout << value1 << " => " << value2 << endl;
          // restore previous behavior
          unbecome();
        }
      );
    }
  );
}
```

An event-based actor finishes execution with normal exit reason if the behavior stack is empty after calling `unbecome`. The default policy of `become` is `discard_behavior` that causes an actor to override its current behavior. The policy flag must be the first argument of `become`.

Note: the message handling in *libcppa* is consistent among all actor implementations: unmatched messages are *never* implicitly discarded if no suitable handler was found. Hence, the order of arrival is not important in the example above. This is unlike other event-based implementations of the actor model such as Akka for instance.

6.3 Timeouts

A behavior set by `become` is invoked whenever a new messages arrives. If no message ever arrives, the actor would wait forever. This might be desirable if the actor only provides a service and should not do anything else. But often, we need to be able to recover if an expected messages does not arrive within a certain time period. The following examples illustrates the usage of `after` to define a timeout.

```
#include <chrono>
#include <iostream>
#include "cppa/cppa.hpp"

using std::endl;

void eager_actor() {
  become (
    on_arg_match >> [](int i) { /* ... */ },
    on_arg_match >> [](float i) { /* ... */ },
    others() >> [] { /* ... */ },
    after(std::chrono::seconds(10)) >> []() {
      aout << "received nothing within 10 seconds..." << endl;
      // ...
    }
  );
}
```

Callbacks given as timeout handler must have zero arguments. Any number of patterns can precede the timeout definition, but "`after`" must always be the final statement. Using a zero-duration timeout causes the actor to scan its mailbox once and then invoke the timeout immediately if no matching message was found.

libcppa supports timeouts using `minutes`, `seconds`, `milliseconds` and `microseconds`. However, note that the precision depends on the operating system and your local work load. Thus, you should not depend on a certain clock resolution.

6.4 Skipping Messages

Unmatched messages are skipped automatically by *libcppa*'s runtime system. This is true for *all* actor implementations. To allow actors to skip messages manually, `skip_message` can be used. This is in particular useful whenever an actor switches between behaviors, but wants to use a default rule created by `others()` to filter messages that are not handled by any of its behaviors.

The following example illustrates a simple server actor that dispatches requests to workers. After receiving an `'idle'` message, it awaits a request that is then forwarded to the idle worker. Afterwards, the server returns to its initial behavior, i.e., awaits the next `'idle'` message. The server actor will exit for reason `user_defined` whenever it receives a message that is neither a request, nor an idle message.

```
void server() {
  auto die = [=] { self->quit(exit_reason::user_defined); };
  become (
    on(atom("idle")) >> [=] {
      auto worker = last_sender();
      become (
        keep_behavior,
        on(atom("request")) >> [=] {
          // forward request to idle worker
          forward_to(worker);
          // await next idle message
          unbecome();
        },
        on(atom("idle")) >> skip_message,
        others() >> die
      );
    },
    on(atom("request")) >> skip_message,
    others() >> die
  );
}
```

7 Synchronous Communication

libcppa uses a future-based API for synchronous communication. The functions `sync_send` and `sync_send_tuple` send synchronous request messages to the receiver and return a future to the response message. Note that the returned future is *actor-local*, i.e., only the actor that has send the corresponding request message is able to receive the response identified by such a future.

```
template<typename... Args>
message_future sync_send(actor_ptr whom, Args&&... what);

message_future sync_send_tuple(actor_ptr whom, any_tuple what);

template<typename Duration, typename... Args>
message_future timed_sync_send(actor_ptr whom,
                               Duration timeout,
                               Args&&... what);

template<typename Duration, typename... Args>
message_future timed_sync_send_tuple(actor_ptr whom,
                                     Duration timeout,
                                     any_tuple what);
```

A synchronous message is sent to the receiving actor's mailbox like any other asynchronous message. The response message, on the other hand, is treated separately.

The difference between `sync_send` and `timed_sync_send` is how timeouts are handled. The behavior of `sync_send` is analogous to `send`, i.e., timeouts are specified by using `after(...)` statements (see 6.3). When using `timed_sync_send` function, `after(...)` statements are ignored and the actor will receive a `'TIMEOUT'` message after the given duration instead.

7.1 Error Messages

When using synchronous messaging, *libcppa*'s runtime environment will send ...

- `{'EXITED', uint32_t exit_reason}` if the receiver is not alive
- `{'VOID'}` if the receiver handled the message but did not respond to it
- `{'TIMEOUT'}` if a message send by `timed_sync_send` timed out

7.2 Receive Response Messages

The function `handle_response` can be used to set a one-shot handler receiving the response message send by `sync_send`.

```
actor_ptr testee = ...; // replies with a string to 'get'

// "handle_response" usage example
auto handle = sync_send(testee, atom("get"));
handle_response (handle) (
  on_arg_match >> [=](const std::string& str) {
    // handle str
  },
  after(std::chrono::seconds(30)) >> [=]() {
    // handle error
  }
);
```

Similar to `become`, the function `handle_response` modifies an actor's behavior stack. However, it is used as "one-shot handler" and automatically returns the previous actor behavior afterwards. It is possible to "stack" multiple `handle_response` calls. Each response handler is executed once and then automatically discarded.

7.3 Synchronous Failures and Error Handlers

An unexpected response message, i.e., a message that is not handled by given behavior, will invoke the actor's `on_sync_failure` handler. The default handler kills the actor by calling `self->quit(exit_reason::unhandled_sync_failure)`. The handler can be overridden by calling `self->on_sync_failure(/*...*/)`.

Unhandled `'TIMEOUT'` messages trigger the `on_sync_timeout` handler. The default handler kills the actor for reason `exit_reason::unhandled_sync_failure`. It is possible set both error handlers by calling `self->on_sync_timeout_or_failure(/*...*)`.

7.4 Using `then` to Receive a Response

Often, an actor sends a synchronous message and then wants to wait for the response. In this case, using either `handle_response` is quite verbose. To allow for a more compact code, `message_future` provides the member function `then`. Using this member function is equal to using `handle_response`, as illustrated by the following example.

```
actor_ptr testee = ...; // replies with a string to 'get'

// set handler for unexpected messages
self->on_sync_failure = [] {
    aout << "received: " << to_string(self->last_dequeued()) << endl;
};

// set handler for timeouts
self->on_sync_timeout = [] {
    aout << "timeout occured" << endl;
};

// set response handler by using "then"
timed_sync_send(testee, std::chrono::seconds(30), atom("get")).then(
  on_arg_match >> [=](const std::string& str) { /* handle str */ }
);
```

7.4.1 Using Functors without Patterns

To reduce verbosity, *libcppa* supports synchronous response handlers without patterns. In this case, the pattern is automatically deduced by the functor's signature.

```
actor_ptr testee = ...; // replies with a string to 'get'

// (1) functor only usage
sync_send(testee, atom("get")).then(
  [=](const std::string& str) { /*...*/ }
);
// statement (1) is equal to:
sync_send(testee, atom("get")).then(
  on(any_vals, arg_match) >> [=](const std::string& str) { /*...*/ }
);
```

7.4.2 Continuations for Event-based Actors

libcppa supports continuations to enable chaining of send/receive statements. The functions `handle_response` and `message_future::then` both return a helper object offering the member function `continue_with`, which takes a functor f without arguments. After receiving a message, f is invoked if and only if the received messages was handled successfully, i.e., neither `sync_failure` nor `sync_timeout` occurred.

```
actor_ptr d_or_s = ...; // replies with either a double or a string

sync_send(d_or_s, atom("get")).then(
  [=](double value) { /* functor f1 */ },
  [=](const string& value) { /* functor f2*/ }
).continue_with([=] {
  // this continuation is invoked in both cases
  // *after* f1 or f2 is done, but *not* in case
  // of sync_failure or sync_timeout
});
```

8 Management & Error Detection

libcppa adapts Erlang's well-established fault propagation model. It allows to build actor subsystem in which either all actors are alive or have collectively failed.

8.1 Links

Linked actors monitor each other. An actor sends an exit message to all of its links as part of its termination. The default behavior for actors receiving such an exit message is to die for the same reason, if the exit reason is non-normal. Actors can *trap* exit messages to handle them manually.

```
actor_ptr worker = ...;
// receive exit messages as regular messages
self->trap_exit(true);
// monitor spawned actor
self->link_to(worker);
// wait until worker exited
become (
  on(atom("EXIT"), exit_reason::normal) >> [] {
    // worker finished computation
  },
  on(atom("EXIT"), arg_match) >> [](std::uint32_t reason) {
    // worker died unexpectedly
  }
);
```

8.2 Monitors

A monitor observes the lifetime of an actor. Monitored actors send a down message to all observers as part of their termination. Unlike exit messages, down messages are always treated like any other ordinary message. An actor will receive one down message for each time it called `self->monitor(...)`, even if it adds a monitor to the same actor multiple times.

```
actor_ptr worker = ...;
// monitor spawned actor
self->monitor(worker);
// wait until worker exited
receive (
  on(atom("DOWN"), exit_reason::normal) >> [] {
    // worker finished computation
  },
  on(atom("DOWN"), arg_match) >> [](std::uint32_t reason) {
    // worker died unexpectedly
  }
);
```

8.3 Error Codes

All error codes are defined in the namespace `cppa::exit_reason`. To obtain a string representation of an error code, use `cppa::exit_reason::as_string(uint32_t)`.

`normal`	1	Actor finished execution without error
`unhandled_exception`	2	Actor was killed due to an unhandled exception
`unallowed_function_call`	3	Indicates that an event-based actor tried to use blocking receive calls
`unhandled_sync_failure`	4	Actor was killed due to an unexpected synchronous response message
`unhandled_sync_timeout`	5	Actor was killed, because no timeout handler was set and a synchronous message timed out
`user_shutdown`	16	Actor was killed by a user-generated event
`remote_link_unreachable`	257	Indicates that a remote actor became unreachable, e.g., due to connection error
`user_defined`	65536	Minimum value for user-defined exit codes

8.4 Attach Cleanup Code to an Actor

Actors can attach cleanup code to other actors. This code is executed immediately if the actor has already exited. Keep in mind that `self` refers to the currently running actor. Thus, `self` refers to the terminating actor and not to the actor that attached a functor to it.

```
auto worker = spawn(...);
actor_ptr observer = self;
// "monitor" spawned actor
worker->attach_functor([observer](std::uint32_t reason) {
  // this callback is invoked from worker
  send(observer, atom("DONE"));
});
// wait until worker exited
become (
  on(atom("DONE")) >> [] {
    // worker terminated
  }
);
```

Note: It is possible to attach code to remote actors, but the cleanup code will run on the local machine.

9 Spawning Actors

Actors are created using the function spawn. The easiest way to implement actors is to use functors, e.g., a free function or lambda expression. The arguments to the functor are passed to spawn as additional arguments. The function spawn also takes optional flags as template paremeter. The flag detached causes spawn to create a thread-mapped actor (opt-out of the cooperative scheduling), the flag linked links the newly created actor to its parent, and the flag monitored automatically adds a monitor to the new actor. Actors that make use of the blocking API (see Section 14) must be spawned using the flag blocking_api. Flags are concatenated using the operator +, as shown in the examples below.

```
#include "cppa/cppa.hpp"

using namespace cppa;

void my_actor1();
void my_actor2(int arg1, const std::string& arg2);
void ugly_duckling();

class my_actor3 : public event_based_actor { /* ... */ };
class my_actor4 : public sb_actor<my_actor4> {
  public: my_actor4(int some_value) { /* ... */ }
  /* ... */
};

int main() {
  // spawn function-based actors
  auto a0 = spawn(my_actor1);
  auto a1 = spawn<linked>(my_actor2, 42, "hello actor");
  auto a2 = spawn<monitored>([] { /* ... */ });
  auto a3 = spawn([](int) { /* ... */ }, 42);
  // spawn thread-mapped actors
  auto a4 = spawn<detached>(my_actor1);
  auto a5 = spawn<detached + linked>([] { /* ... */ });
  auto a6 = spawn<detached>(my_actor2, 0, "zero");
  // spawn class-based actors
  auto a7 = spawn<my_actor3>();
  auto a8 = spawn<my_actor4, monitored>(42);
  // spawn thread-mapped actors using a class
  auto a9 = spawn<my_actor4, detached>(42);
  // spawn actors that need access to the blocking API
  auto b0 = spawn<blocking_api>(ugly_duckling);
}
```

Note: spawn(fun, arg0, ...) is **not** equal to spawn(std::bind(fun, arg0, ...))! For example, a call to spawn(fun, self, ...) will pass a pointer to the calling actor to the newly created actor, as expected, whereas spawn(std::bind(fun, self, ...)) wraps the type of self into the function wrapper and evaluates self on function invocation. Thus, the actor will end up having a pointer *to itself* rather than a pointer to its parent.

10 Message Priorities

By default, all messages have the same priority and actors ignore priority flags. Actors that should evaluate priorities must be spawned using the `priority_aware` flag. This flag causes the actor to use a priority-aware mailbox implementation. It is not possible to change this implementation dynamically at runtime.

```
void testee() {
  // send 'b' with normal priority
  send(self, atom("b"));
  // send 'a' with high priority
  send({self, message_priority::high}, atom("a"));
  // terminate after receiving a 'b'
  become (
    on(atom("b")) >> [] {
      aout << "received 'b' => quit" << endl;
      self->quit();
    },
    on(atom("a")) >> [] {
      aout << "received 'a'" << endl;
    },
  );
}

int main() {
  // will print "received 'b' => quit"
  spawn(testee);
  await_all_others_done();
  // will print "received 'a'" and then "received 'b' => quit"
  spawn<priority_aware>(testee);
  await_all_others_done();
  shutdown();
}
```

11 Network Transparency

All actor operations as well as sending messages are network transparent. Remote actors are represented by actor proxies that forward all messages.

11.1 Publishing of Actors

```
void publish(actor_ptr whom, std::uint16_t port, const char* addr = 0)
```

The function `publish` binds an actor to a given port. It throws `network_error` if socket related errors occur or `bind_failure` if the specified port is already in use. The optional `addr` parameter can be used to listen only to the given IP address. Otherwise, the actor accepts all incoming connections (`INADDR_ANY`).

```
publish(self, 4242);
become (
  on(atom("ping"), arg_match) >> [](int i) {
    return make_cow_tuple(atom("pong"), i);
  }
);
```

11.2 Connecting to Remote Actors

```
actor_ptr remote_actor(const char* host, std::uint16_t port)
```

The function `remote_actor` connects to the actor at given host and port. A `network_error` is thrown if the connection failed.

```
auto pong = remote_actor("localhost", 4242);
send(pong, atom("ping"), 0);
become (
  on(atom("pong"), 10) >> [] {
    self->quit();
  },
  on(atom("pong"), arg_match) >> [](int i) {
    return make_cow_tuple(atom("ping"), i+1);
  }
);
```

12 Group Communication

libcppa supports publish/subscribe-based group communication. Actors can join and leave groups and send messages to groups.

```
std::string group_module = ...;
std::string group_id = ...;
auto grp = group::get(group_module, group_id);
self->join(grp);
send(grp, atom("test"));
self->leave(grp);
```

12.1 Anonymous Groups

Groups created on-the-fly with `group::anonymous()` can be used to coordinate a set of workers. Each call to `group::anonymous()` returns a new, unique group instance.

12.2 Local Groups

The `"local"` group module creates groups for in-process communication. For example, a group for GUI related events could be identified by `group::get("local", "GUI events")`. The group ID `"GUI events"` uniquely identifies a singleton group instance of the module `"local"`.

12.3 Spawn Actors in Groups

The function `spawn_in_group` can be used to create actors as members of a group. The function causes the newly created actors to call `self->join(...)` immediately and before `spawn_in_group` returns. The usage of `spawn_in_group` is equal to `spawn`, except for an additional group argument. The group handle is always the first argument, as shown in the examples below.

```
void fun1();
void fun2(int, float);
class my_actor1 : event_based_actor { /* ... */ };
class my_actor2 : event_based_actor {
  // ...
  my_actor2(const std::string& str) { /* ... */ }
};
// ...
auto grp = group::get(...);
auto a1 = spawn_in_group(grp, fun1);
auto a2 = spawn_in_group(grp, fun2, 1, 2.0f);
auto a3 = spawn_in_group<my_actor1>(grp);
auto a4 = spawn_in_group<my_actor2>(grp, "hello my_actor2!");
```

13 Platform-Independent Type System

libcppa provides a fully network transparent communication between actors. Thus, *libcppa* needs to serialize and deserialize messages. Unfortunately, this is not possible using the RTTI system of C++. *libcppa* uses its own RTTI based on the class `uniform_type_info`, since it is not possible to extend `std::type_info`.

Unlike `std::type_info::name()`, `uniform_type_info::name()` is guaranteed to return the same name on all supported platforms. Furthermore, it allows to create an instance of a type by name.

```
// creates a signed, 32 bit integer
cppa::object i = cppa::uniform_typeid<int>()->create();
```

However, you should rarely if ever need to use `object` or `uniform_type_info`.

13.1 User-Defined Data Types in Messages

All user-defined types must be explicitly "announced" so that *libcppa* can (de)serialize them correctly, as shown in the example below.

```
#include "cppa/cppa.hpp"
using namespace cppa;

struct foo { int a; int b; };

int main() {
  announce<foo>(&foo::a, &foo::b);
  send(self, foo{1,2});
  return 0;
}
```

Without the `announce` function call, the example program would terminate with an exception, because *libcppa* rejects all types without available runtime type information.

`announce()` takes the class as template parameter and pointers to all members (or getter/setter pairs) as arguments. This works for all primitive data types and STL compliant containers. See the announce examples 1 – 4 of the standard distribution for more details.

Obviously, there are limitations. You have to implement serialize/deserialize by yourself if your class does implement an unsupported data structure. See `announce_example_5.cpp` in the examples folder.

14 Blocking API

Besides event-based actors (the default implementation), *libcppa* also provides context-switching and thread-mapped actors that can make use of the blocking API. Those actor implementations are intended to ease migration of existing applications or to implement actors that need to have access to blocking receive primitives for other reasons.

Event-based actors differ in receiving messages from context-switching and thread-mapped actors: the former define their behavior as a message handler that is invoked whenever a new messages arrives in the actor's mailbox (by using `become`), whereas the latter use an explicit, blocking receive function.

14.1 Receiving Messages

The function `receive` sequentially iterates over all elements in the mailbox beginning with the first. It takes a partial function that is applied to the elements in the mailbox until an element was matched by the partial function. An actor calling `receive` is blocked until it successfully dequeued a message from its mailbox or an optional timeout occurs.

```
receive (
  on<int>().when(_x1 > 0) >> // ...
);
```

The code snippet above illustrates the use of `receive`. Note that the partial function passed to `receive` is a temporary object at runtime. Hence, using receive inside a loop would cause creation of a new partial function on each iteration. *libcppa* provides three predefined receive loops to provide a more efficient but yet convenient way of defining receive loops.

```
//DON'T

for (;;) {
  receive (
    // ...
  );
}



std::vector<int> results;
for (size_t i = 0; i < 10; ++i) {
  receive (
    on<int>() >> [&](int value) {
      results.push_back(value);
    }
  );
}



size_t received = 0;
do {
  receive (
    others() >> [&]() {
      ++received;
    }
  );
} while (received < 10);
```

```
//DO

receive_loop (
  // ...
);






std::vector<int> results;
size_t i = 0;
receive_for(i, 10) (
  on<int>() >> [&](int value) {
    results.push_back(value);
  }
);




size_t received = 0;
do_receive (
  others() >> [&]() {
    ++received;
  }
).until(gref(received) >= 10);
```

The examples above illustrate the correct usage of the three loops `receive_loop`, `receive_for` and `do_receive(...).until`. It is possible to nest receives and receive loops.

```
receive_loop (
  on<int>() >> [](int value1) {
    receive (
      on<float>() >> [&](float value2) {
        cout << value1 << " => " << value2 << endl;
      }
    );
  }
);
```

14.2 Receiving Synchronous Responses

```
actor_ptr testee = ...; // replies with a string to 'get'

auto future = sync_send(testee, atom("get"));
receive_response (future) (
  on_arg_match >> [&](const std::string& str) {
    // handle str
  },
  after(std::chrono::seconds(30)) >> [&]() {
    // handle error
  }
);

// or:

sync_send(testee, atom("get")).await (
  on_arg_match >> [&](const std::string& str) {
    // handle str
  },
  after(std::chrono::seconds(30)) >> [&]() {
    // handle error
  }
);
```

15 Strongly Typed Actors

Strongly typed actors provide a convenient way of defining type-safe messaging interfaces. Unlike “dynamic actors”, typed actors are not allowed to change their behavior at runtime, neither are typed actors allowed to use guard expressions. When calling `become` in a strongly typed actor, the actor will be killed with exit reason `unallowed_function_call`.

Typed actors use `typed_actor_ptr<...>` instead of `actor_ptr`, whereas the template parameters hold the messaging interface. For example, an actor responding to two integers with a dobule would use the type `typed_actor_ptr<replies_to<int, int>::with<double>>`.

All functions for message passing, linking and monitoring are overloaded to accept both types of actors. As of version 0.8, strongly typed actors cannot be published and do not support message priorities (those are planned feature for future releases).

15.1 Spawning Typed Actors

Actors are spawned using the function `spawn_typed`. The argument to this function call *must* be a match expression as shown in the example below, because the runtime of libcppa needs to evaluate the signature of each message handler.

```
auto p0 = spawn_typed(
  on_arg_match >> [](int a, int b) {
   return static_cast<double>(a) * b;
  },
  on_arg_match >> [](double a, double b) {
    return make_cow_tuple(a * b, a / b);
  }
);
// assign to identical type
using full_type = typed_actor_ptr<
                    replies_to<int, int>::with<double>,
                    replies_to<double, double>::with<double, double>
                  >;
full_type p1 = p0;
// assign to subtype
using subtype1 = typed_actor_ptr<
                   replies_to<int, int>::with<double>
                 >;
subtype1 p2 = p0;
// assign to another subtype
using subtype2 = typed_actor_ptr<
                   replies_to<double, double>::with<double, double>
                 >;
subtype2 p3 = p0;
```

15.2 Class-based Typed Actors

Typed actors can be implemented using a class by inheriting from `typed_actor<...>`, whereas the template parameter pack denotes the messaging interface. Derived classes have to implemented the pure virtual member function `make_behavior`, as shown in the example below.

```
// implementation
class typed_testee : public typed_actor<replies_to<int>::with<bool>> {

 protected:

  behavior_type make_behavior() final {
    // returning a non-matching expression
    // results in a compile-time error
    return (
      on_arg_match >> [](int value) {
        return value == 42;
      }
    );
  }

};

// instantiation
auto testee = spawn_typed<typed_testee>();
```

It is worth mentioning that `typed_actor` implements the member function `init()` using the `final` qualifier. Hence, derived classes are not allowed to override `init()`. However, typed actors are allowed to override other member functions such as `on_exit()`. The return type of `make_behavior` is `typed_behavior<...>`, which is aliased as `behavior_type` for convenience.

16 Common Pitfalls

16.1 Event-Based API

- The functions `become` and `handle_response` do not block, i.e., always return immediately. Thus, you should *always* capture by value in event-based actors, because all references on the stack will cause undefined behavior if a lambda is executed.

16.2 Mixing Event-Based and Blocking API

- Blocking *libcppa* function such as `receive` will *throw an exception* if accessed from an event-based actor.
- Context-switching and thread-mapped actors *can* use the `become` API. Whenever a non-event-based actor calls `become()` for the first time, it will create a behavior stack and execute it until the behavior stack is empty. Thus, the *initial* `become` blocks until the behavior stack is empty, whereas all subsequent calls to `become` will return immediately. Related functions, e.g., `sync_send(...).then(...)`, behave the same, as they manipulate the behavior stack as well.

16.3 Synchronous Messages

- `send(self->last_sender(), ...)` does **not** send a response message.
- A handle returned by `sync_send` represents *exactly one* response message. Therefore, it is not possible to receive more than one response message.
- The future returned by `sync_send` is bound to the calling actor. It is not possible to transfer such a future to another actor. Calling `receive_response` or `handle_response` for a future bound to another actor is undefined behavior.

16.4 Sending Messages

- `send(whom, ...)` is syntactic sugar for `whom << make_any_tuple(...)`. Hence, a message sent via `send(whom, self->last_dequeued())` will not yield the expected result, since it wraps `self->last_dequeued()` into another `any_tuple` instance. The correct way of forwarding messages is `self->forward_to(whom)`.

16.5 Sharing

- It is strongly recommended to **not** share states between actors. In particular, no actor shall ever access member variables or member functions of another actor. Accessing shared memory segments concurrently can cause undefined behavior that is incredibly hard to find and debug. However, sharing *data* between actors is fine, as long as the data is *immutable* and all actors access the data only via smart pointers such as `std::shared_ptr`. Nevertheless, the recommended way of sharing informations is message passing. Sending data to multiple actors does *not* result in copying the data several times. Read Section 2 to learn more about *libcppa*'s copy-on-write optimization for tuples.

16.6 Constructors of Class-based Actors

- During constructor invocation, `self` does **not** point to `this`. It points to the invoking actor instead.
- You should **not** send or receive messages in a constructor or destructor.

17 Appendix

17.1 Class `option`

Defined in header `"cppa/option.hpp"`.

```
template<typename T>
class option;
```

Represents an optional value.

Member types

Member type	Definition
`type`	`T`

Member functions

`option()`	Constructs an empty option
`option(T value)`	Initializes `this` with `value`
`option(const option&)` `option(option&&)`	Copy/move construction
`option& operator=(const option&)` `option& operator=(option&&)`	Copy/move assignment

Observers

`bool valid()` `explicit operator bool()`	Returns `true` if `this` has a value
`bool empty()` `bool operator!()`	Returns `true` if `this` does **not** has a value
`const T& get()` `const T& operator*()`	Access stored value
`const T& get_or_else(const T& x)`	Returns `get()` if valid, `x` otherwise

Modifiers

`T& get()` `T& operator*()`	Access stored value

17.2 Using aout – A Concurrency-safe Wrapper for cout

When using `cout` from multiple actors, output often appears interleaved. Moreover, using `cout` from multiple actors – and thus multiple threads – in parallel should be avoided, since the standard does not guarantee a thread-safe implementation.

By replacing `std::cout` with `cppa::aout`, actors can achieve a concurrency-safe text output. The header `cppa/cppa.hpp` also defines overloads for `std::endl` and `std::flush` for `aout`, but does not support the full range of ostream operations (yet). Each write operation to `aout` sends a message to a 'hidden' actor (keep in mind, sending messages from actor constructors is not safe). This actor only prints lines, unless output is forced using `flush`.

```
#include <chrono>
#include <cstdlib>
#include "cppa/cppa.hpp"

using namespace cppa;
using std::endl;

int main() {
    std::srand(std::time(0));
    for (int i = 1; i <= 50; ++i) {
        spawn([i] {
          aout << "Hi there! This is actor nr. " << i << "!" << endl;
          std::chrono::milliseconds tout{std::rand() % 1000};
          delayed_send(self, tout, atom("done"));
          receive(others() >> [i] {
              aout << "Actor nr. " << i << " says goodbye!" << endl;
          });
        });
    }
    // wait until all other actors we've spawned are done
    await_all_others_done();
    // done
    shutdown();
    return 0;
}
```