THEATR

an Actor-Model Language so easy, even an Actor/Model could use it!

Our Team:

Betsy Carroll
Suraj Keshri
Mike Lin
Linda Ortega

"All the GLOBAL SCOPE's a THEATER ™,

And all the INSTANCES of NON PRIMITIVE/NON BUILT IN DATA TYPES

...merely ACTORS."

-William Shakespeare, PLT Spring 1582

The Actor Model

- Actor = the primitive unit of computation
- Actor = sort of like Objects in OO-model languages
    - BUT DIFFERENT!!!

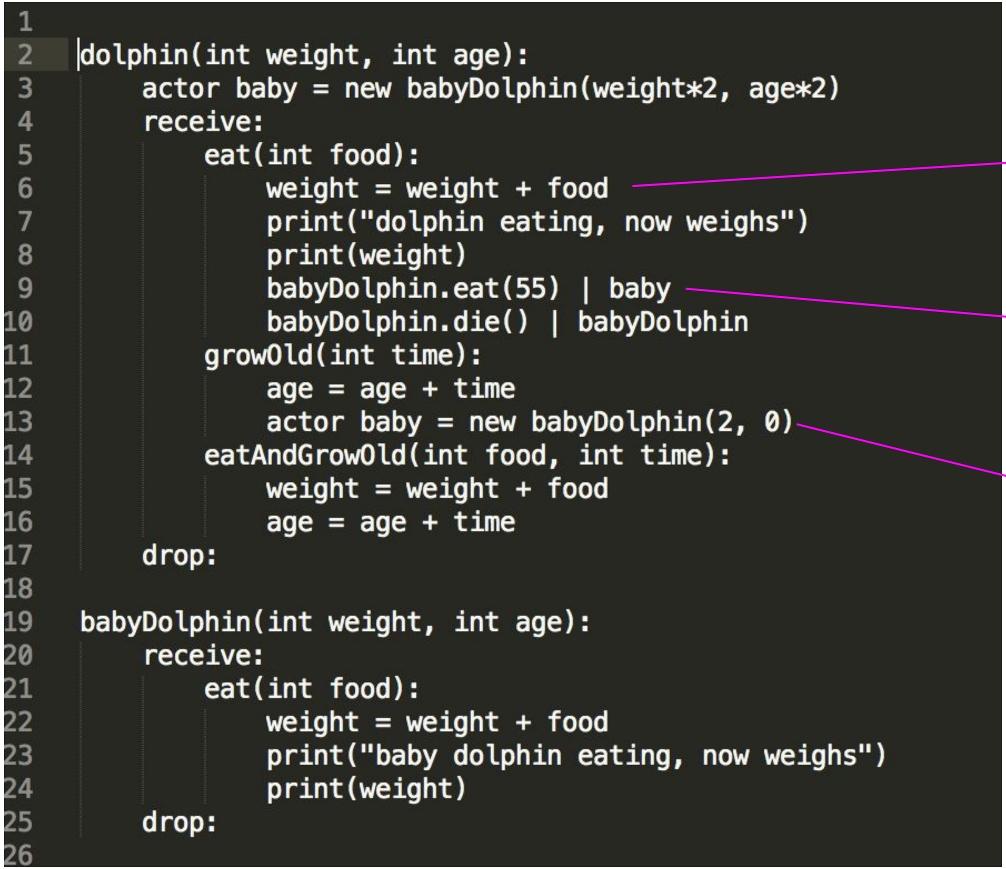
What Actors Can Do

An actor can **hold messages in a queue**

An actor can **dequeue one message**

An actor can do 1 of 3 **things in response to the dequeued message**:

1.) Create more actor(s)
2.) Send message(s) to other actors
3.) Change its internal state (aka designate what it will do with the next message it dequeues



```
dolphin(int weight, int age):
    actor baby = new babyDolphin(weight*2, age*2)
    receive:
        eat(int food):
            weight = weight + food
            print("dolphin eating, now weighs")
            print(weight)
            babyDolphin.eat(55) | baby
            babyDolphin.die() | babyDolphin
        growOld(int time):
            age = age + time
            actor baby = new babyDolphin(2, 0)
        eatAndGrowOld(int food, int time):
            weight = weight + food
            age = age + time
    drop:

babyDolphin(int weight, int age):
    receive:
        eat(int food):
            weight = weight + food
            print("baby dolphin eating, now weighs")
            print(weight)
    drop:
```

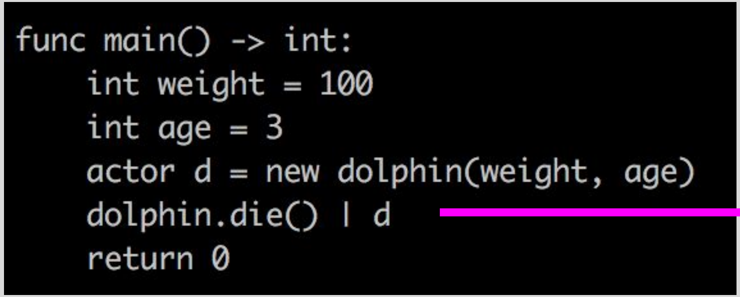
// **upon receiving message:**

// change its internal state (weight)

//send a message to another actor

//create a new actor

Theatr: actors’ methods are in the form of messages



```
func main() -> int:
    int weight = 100
    int age = 3
    actor d = new dolphin(weight, age)
    dolphin.die() | d
    return 0
```

type.please_do_something | instance

// a message is piped thru to an actor instance

// the actor then handles the message and decides what to do in reaction to the request to do something on its own time internally

Actor’s Mailbox = message queue

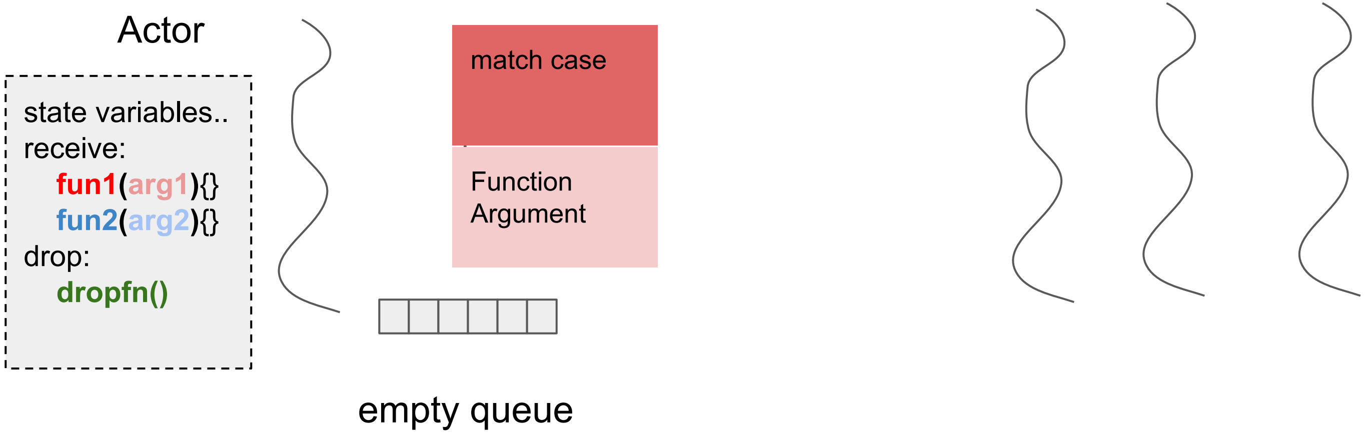
All functions come in the form of a request to do something that is sent to the actor’s **message queue (aka mailbox)**

Although **multiple actors can run at the same time**, an **actor will process** messages **sequentially**

If you send 3 messages to 1 actor, that actor will dequeue them and then process each message one at a time ↣ **asynchronous**

Because of this sequential processing, an actor needs a place to store unprocessed messages as they come in ↣ the message queue.

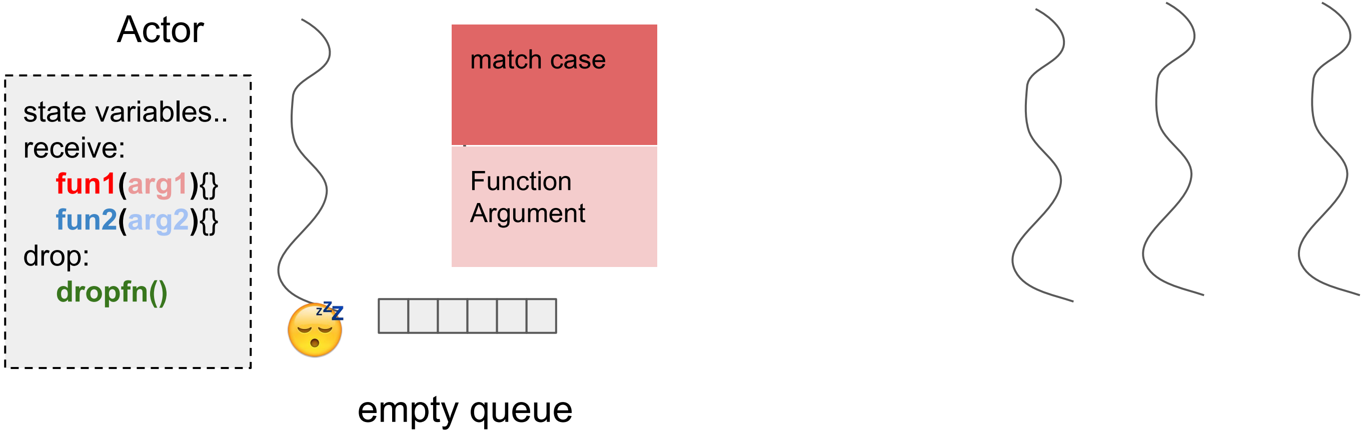
Message Implementation



Actor

```
state variables..
receive:
   fun1(arg1){}
   fun2(arg2){}
drop:
   dropfn()
```

Message Implementation



Actor

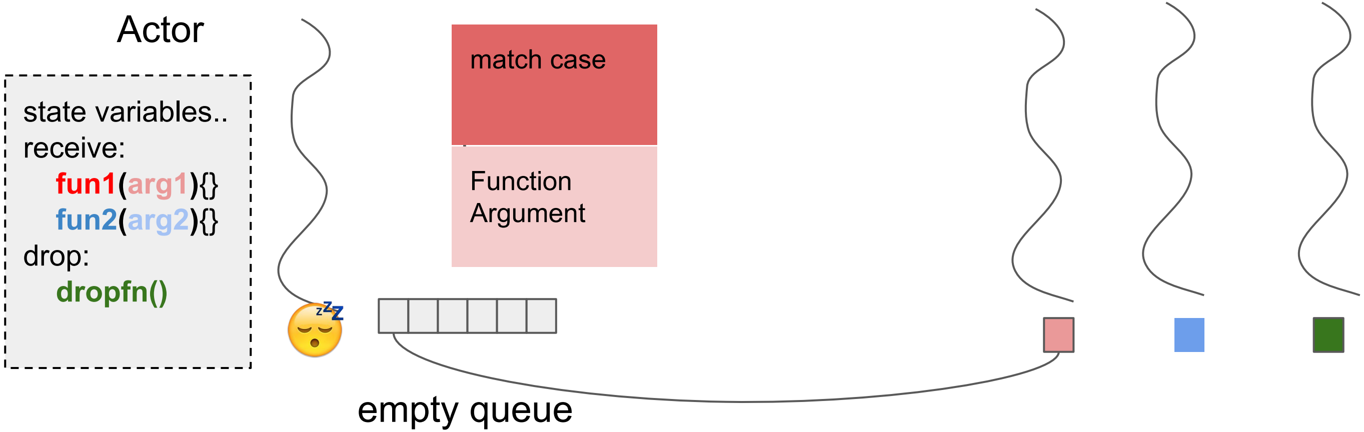
```
state variables..
receive:
    fun1(arg1){}
    fun2(arg2){}
drop:
    dropfn()
```

match case

Function
Argument

empty queue

Message Implementation



Actor

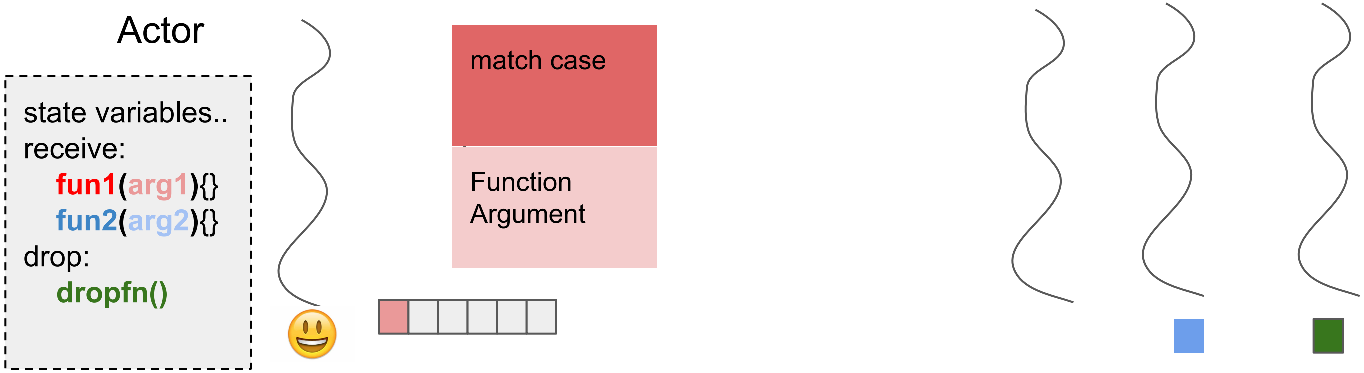
```
state variables..
receive:
    fun1(arg1){}
    fun2(arg2){}
drop:
    dropfn()
```

match case

Function Argument

empty queue

Message Implementation



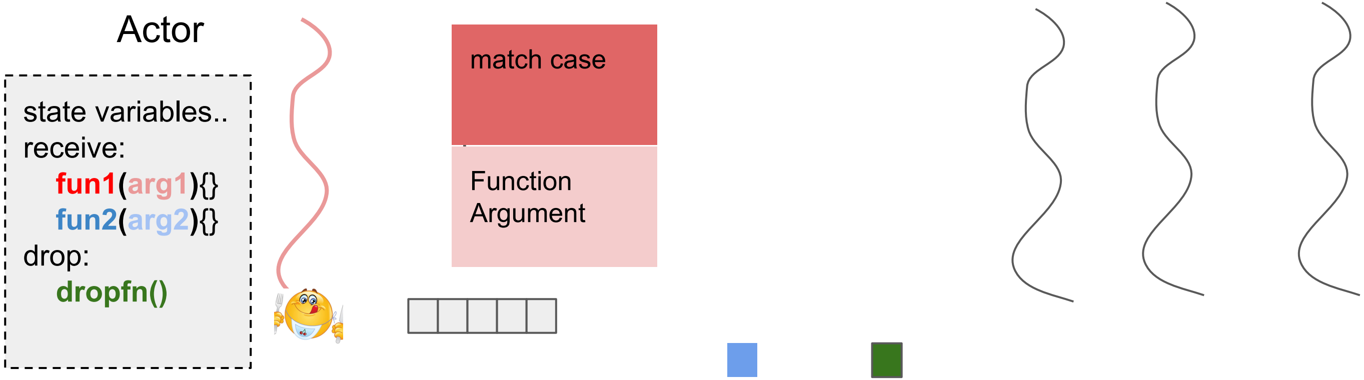
Actor

```
state variables..
receive:
    fun1(arg1){}
    fun2(arg2){}
drop:
    dropfn()
```

match case

Function Argument

Message Implementation



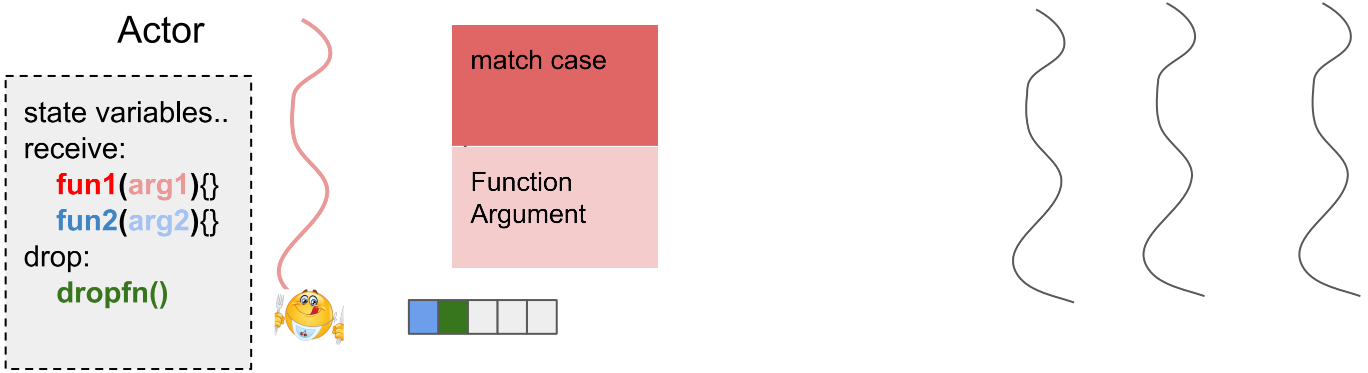
Actor

```
state variables..
receive:
   fun1(arg1){}
   fun2(arg2){}
drop:
   dropfn()
```

match case

Function Argument

Message Implementation



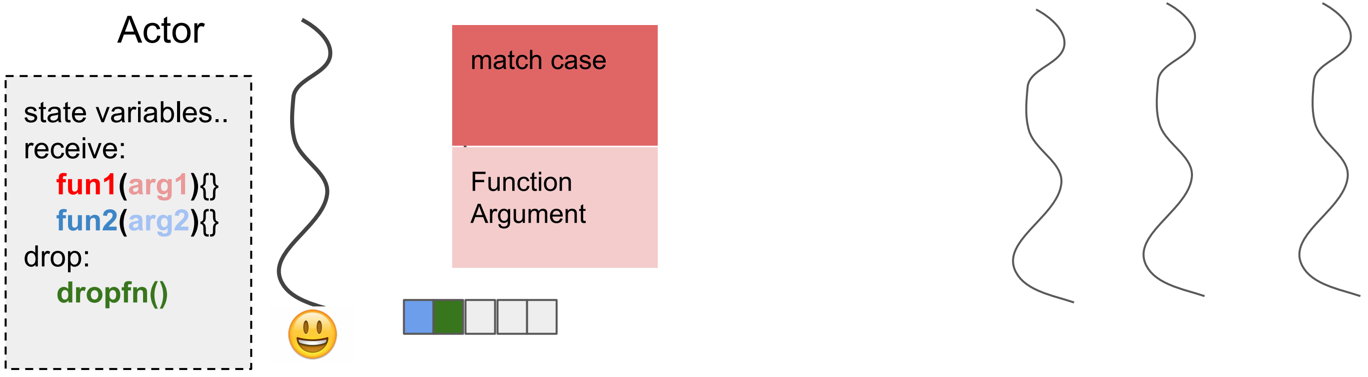
Actor

```
state variables..
receive:
   fun1(arg1){}
   fun2(arg2){}
drop:
   dropfn()
```

match case

Function Argument

Message Implementation



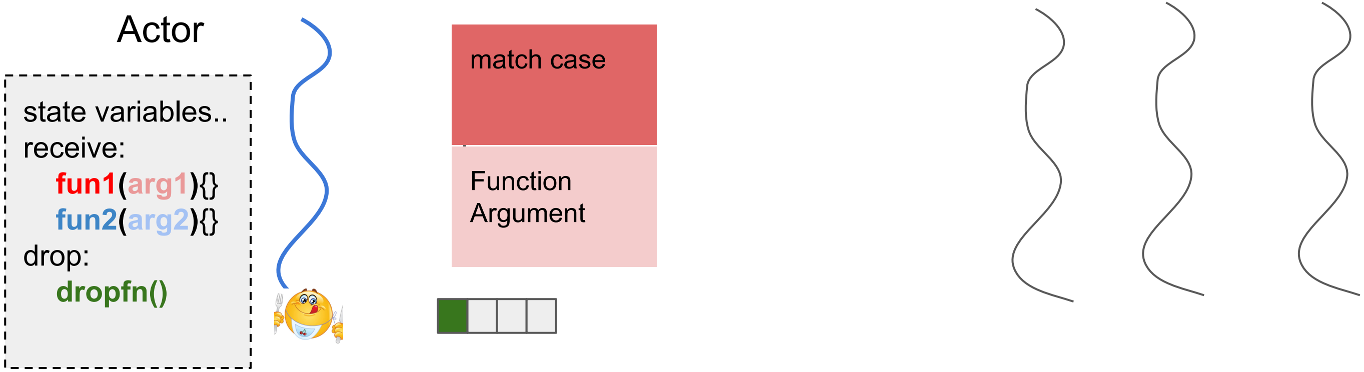
Actor

```
state variables..
receive:
   fun1(arg1){}
   fun2(arg2){}
drop:
   dropfn()
```

match case

Function
Argument

Message Implementation



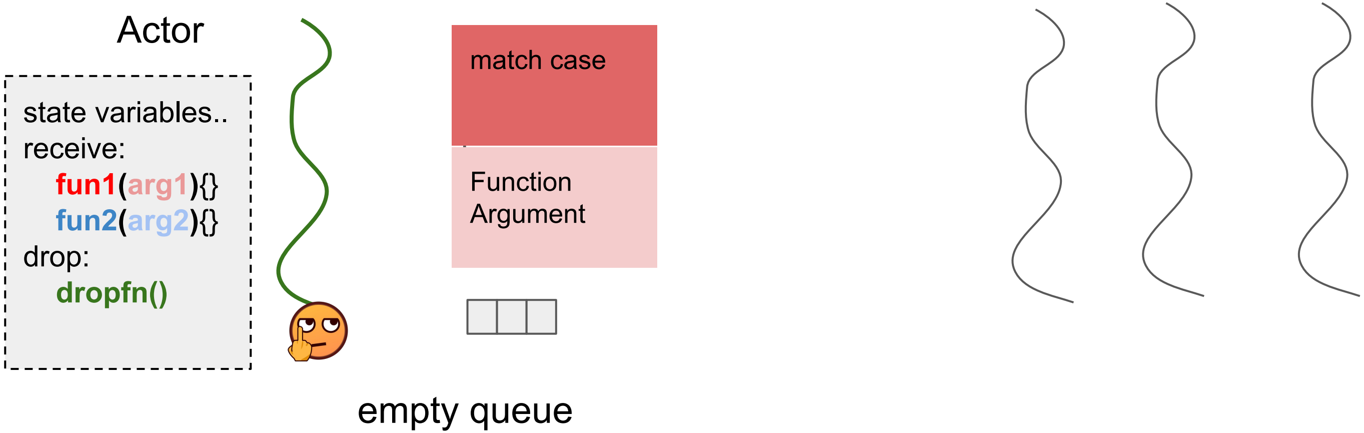
Actor

```
state variables..
receive:
   fun1(arg1){}
   fun2(arg2){}
drop:
   dropfn()
```

match case

Function
Argument

Message Implementation



Actor

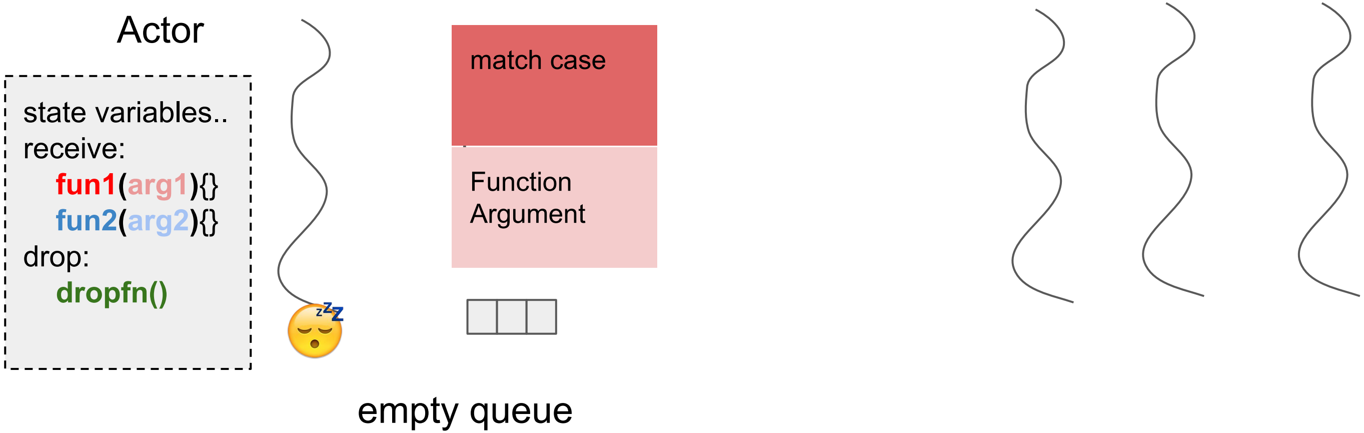
state variables..
receive:
    fun1(arg1){}
    fun2(arg2){}
drop:
    dropfn()

match case

Function
Argument

empty queue

Message Implementation



Actor

```
state variables..
receive:
    fun1(arg1){}
    fun2(arg2){}
drop:
    dropfn()
```

Why use Actor Model?

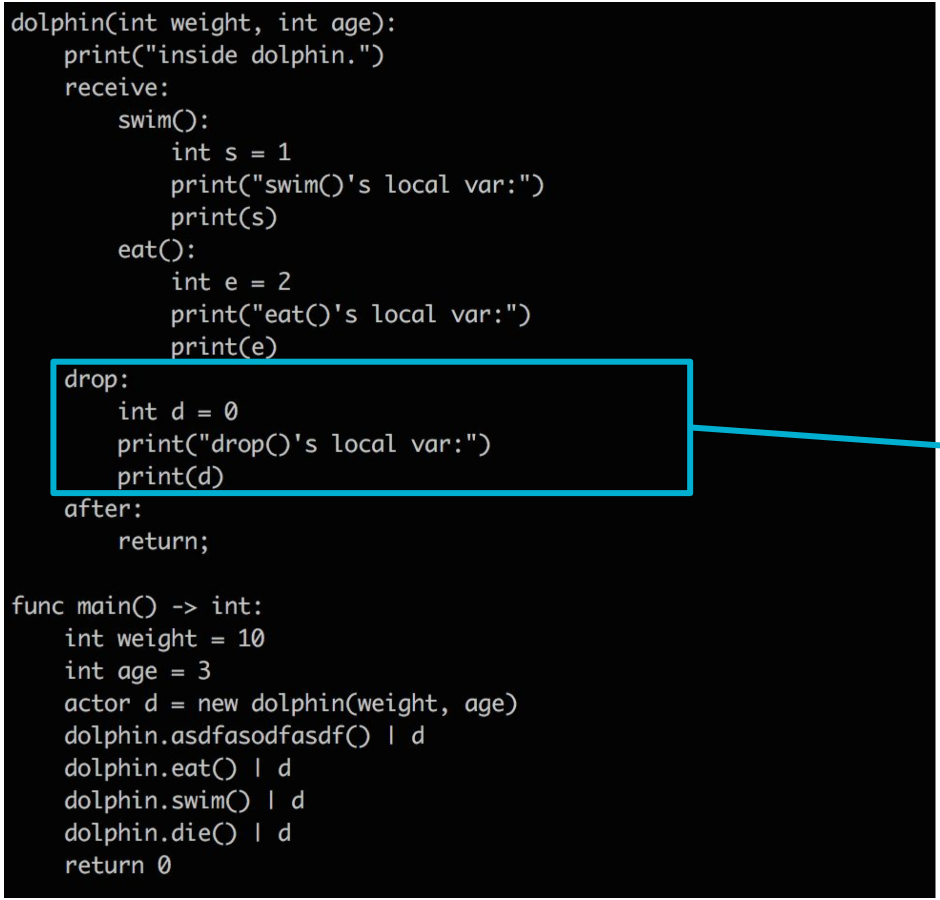
“Let it Crash” Philosophy

The **programmer shouldn’t have to anticipate** and try to account for all possible problems.

Instead: **you should just let it crash (gracefully).**

“Let it Crash”

In THEATR



```
dolphin(int weight, int age):
    print("inside dolphin.")
    receive:
        swim():
            int s = 1
            print("swim()'s local var:")
            print(s)
        eat():
            int e = 2
            print("eat()'s local var:")
            print(e)
    drop:
        int d = 0
        print("drop()'s local var:")
        print(d)
    after:
        return;

func main() -> int:
    int weight = 10
    int age = 3
    actor d = new dolphin(weight, age)
    dolphin.asdfasodfasdf() | d
    dolphin.eat() | d
    dolphin.swim() | d
    dolphin.die() | d
    return 0
```

Drop method

“Let it Crash” Philosophy

Instead: **you should just let it crash (gracefully). Actor model does this well:**

- **actors just drop messages** that they don’t know how to handle.
    - They don’t freak out, they continue to be in the stable state they were in before, the program just moves on.
- **You can make actors whose sole job is to watch the various actors/processes**
    - “One ant is no ant”…. But ants are cheap and so are actors! So you can go wild with em
    - Have **supervisor actors** who watch other actors and and **reset them to stable state if something does crash**

Implementation

From C:

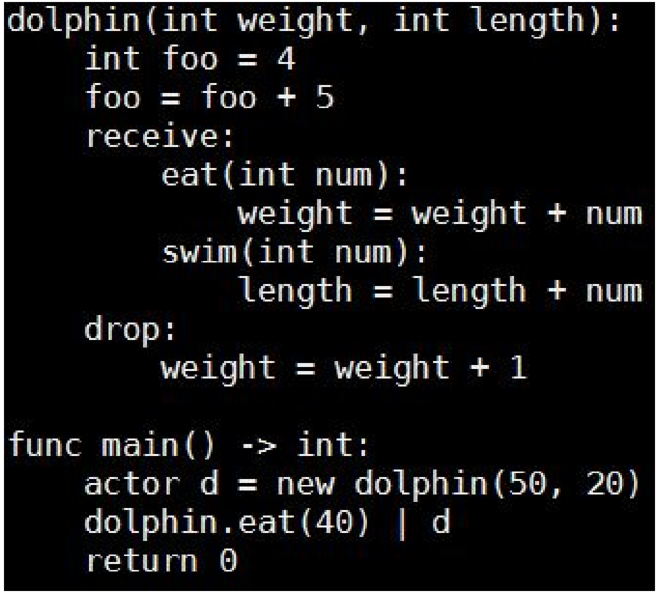
pthread_create

Queue implementation

Mutexes and condition variables

LLVM:

Everything else

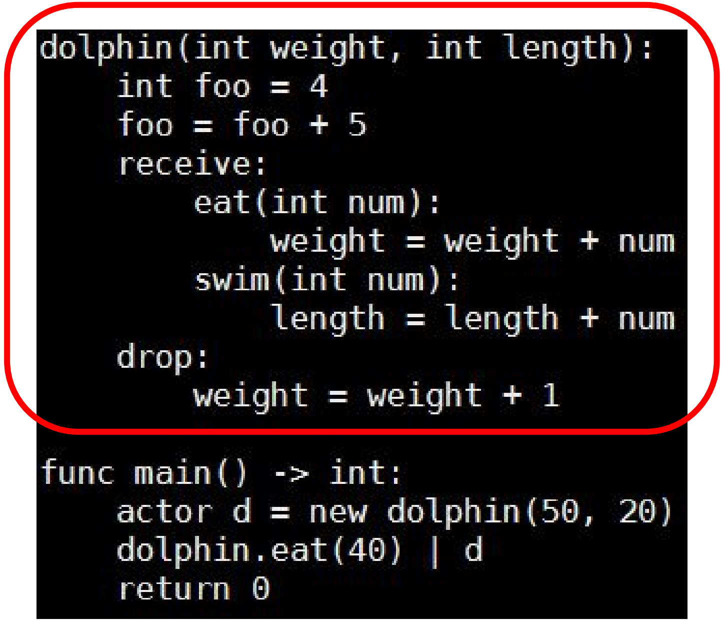


```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1

func main() -> int:
    actor d = new dolphin(50, 20)
    dolphin.eat(40) | d
    return 0
```

Implementation - Actors in Threads

Q: How do we get actors to run independently?

- For each actor declaration, build a function representing these statements to be passed to `pthread_create` whenever a new actor of that type is made



```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1

func main() -> int:
    actor d = new dolphin(50, 20)
    dolphin.eat(40) | d
    return 0
```

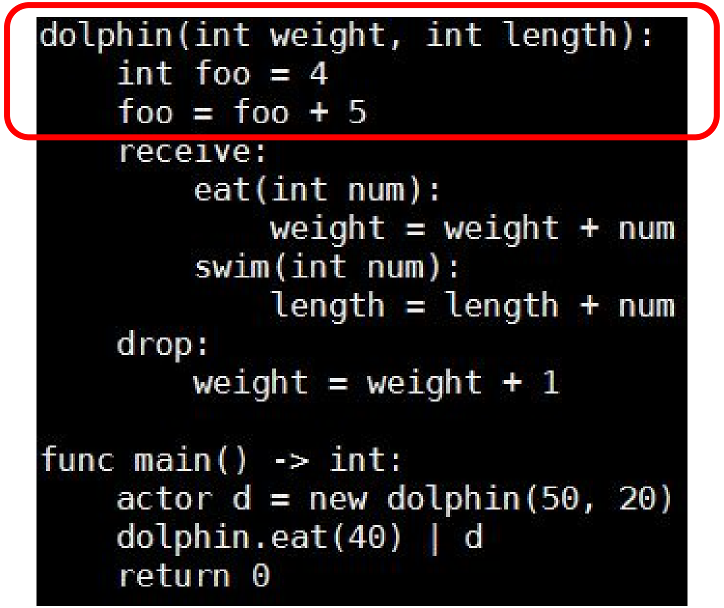
Implementation - Actors in Threads

Q: How do we get actors to run independently?

- For each actor declaration, build a function representing these statements to be passed to `pthread_create` whenever a new actor of that type is made

1) Copy formals and locals onto the stack

2) An invisible argument is a pointer to the message queue that this thread will read from



```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1

func main() -> int:
    actor d = new dolphin(50, 20)
    dolphin.eat(40) | d
    return 0
```

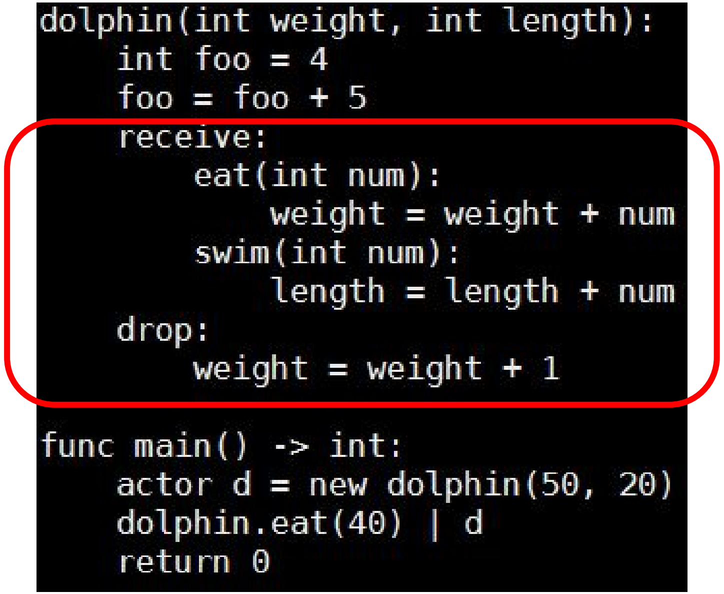
Implementation - Actors in Threads

Q: How do we get actors to run independently?

- For each actor declaration, build a function representing these statements to be passed to `pthread_create` whenever a new actor of that type is made

3) Transform the receive and drop functions into a switch-case block running in an infinite loop.

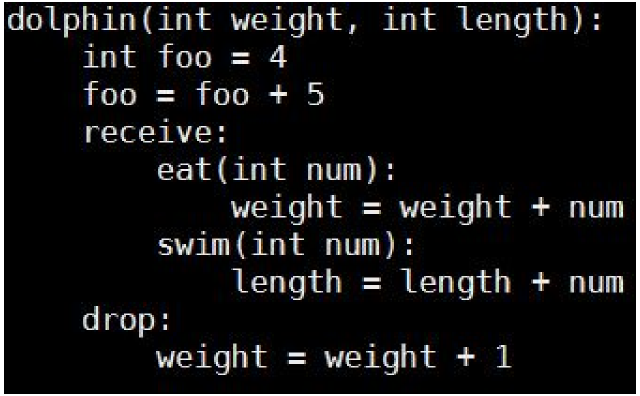
- At each iteration of the loop, a new message is pulled off the queue and the corresponding case statement is called
- A StringMap is built to keep track of function names to case numbers



```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1

func main() -> int:
    actor d = new dolphin(50, 20)
    dolphin.eat(40) | d
    return 0
```

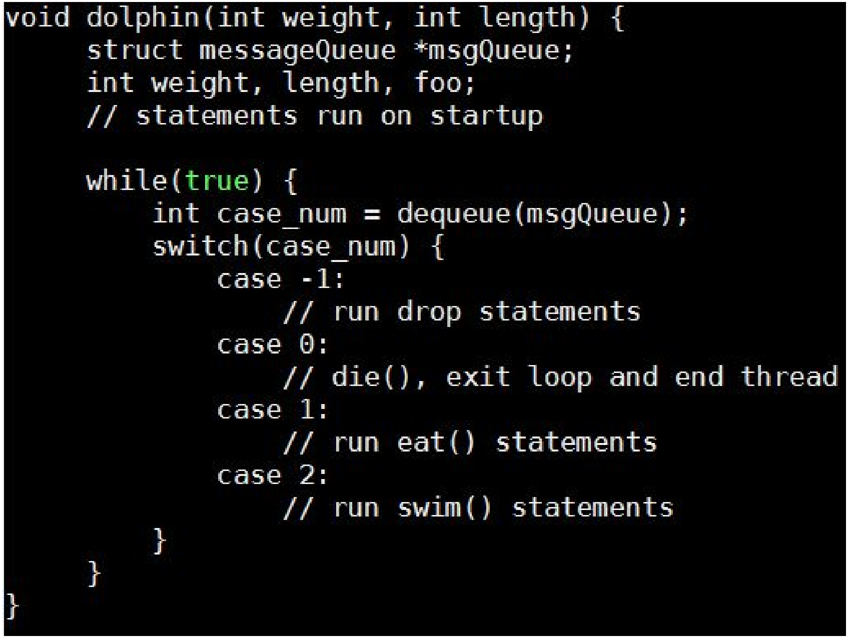
Implementation - Actors in Threads

Theatr code written



```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1
```

Equivalent C-code generated in LLVM

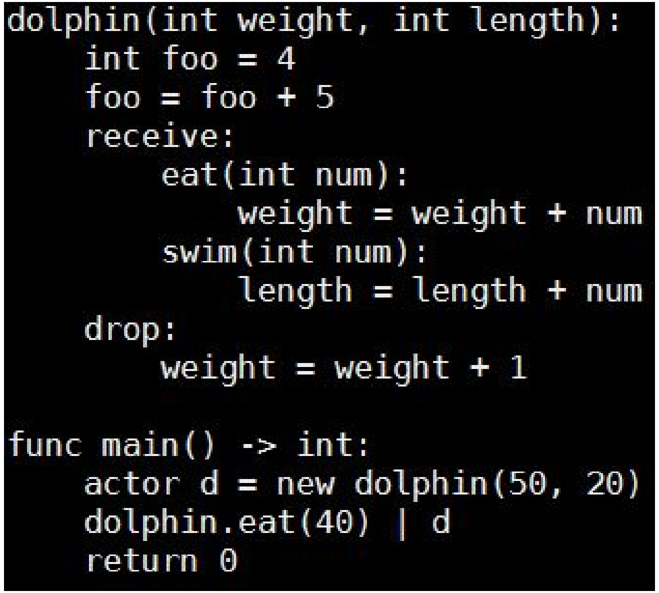


```
void dolphin(int weight, int length) {
    struct messageQueue *msgQueue;
    int weight, length, foo;
    // statements run on startup

    while(true) {
        int case_num = dequeue(msgQueue);
        switch(case_num) {
            case -1:
                // run drop statements
            case 0:
                // die(), exit loop and end thread
            case 1:
                // run eat() statements
            case 2:
                // run swim() statements
        }
    }
}
```

Implementation - Features of Message Statements

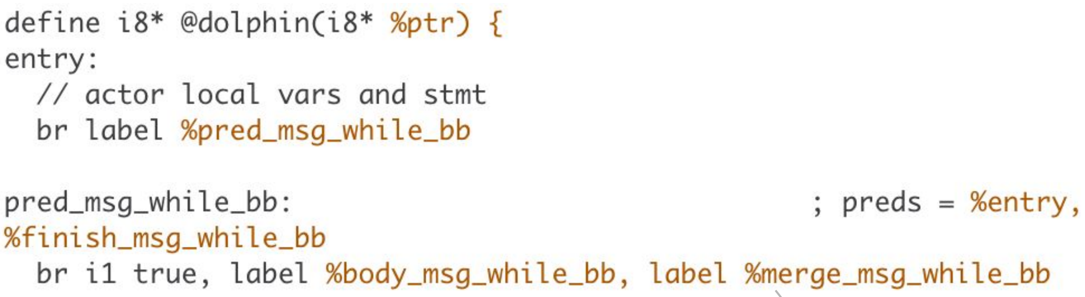
Similar scoping as nested functions.

Associated with a unique case number.

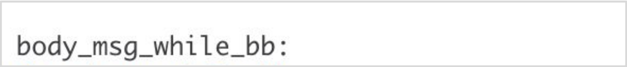


```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1

func main() -> int:
    actor d = new dolphin(50, 20)
    dolphin.eat(40) | d
    return 0
```

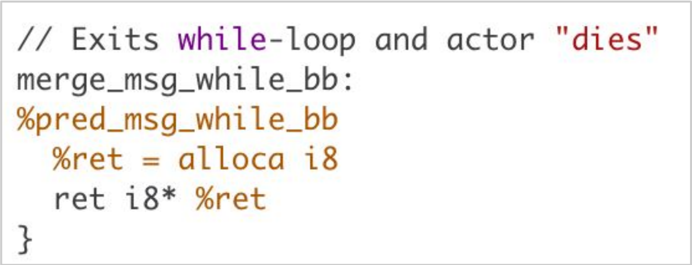
Implementation - Message Cases



```
define i8* @dolphin(i8* %ptr) {
entry:
  // actor local vars and stmt
  br label %pred_msg_while_bb

pred_msg_while_bb:                                              ; preds = %entry,
%finish_msg_while_bb
  br i1 true, label %body_msg_while_bb, label %merge_msg_while_bb
```

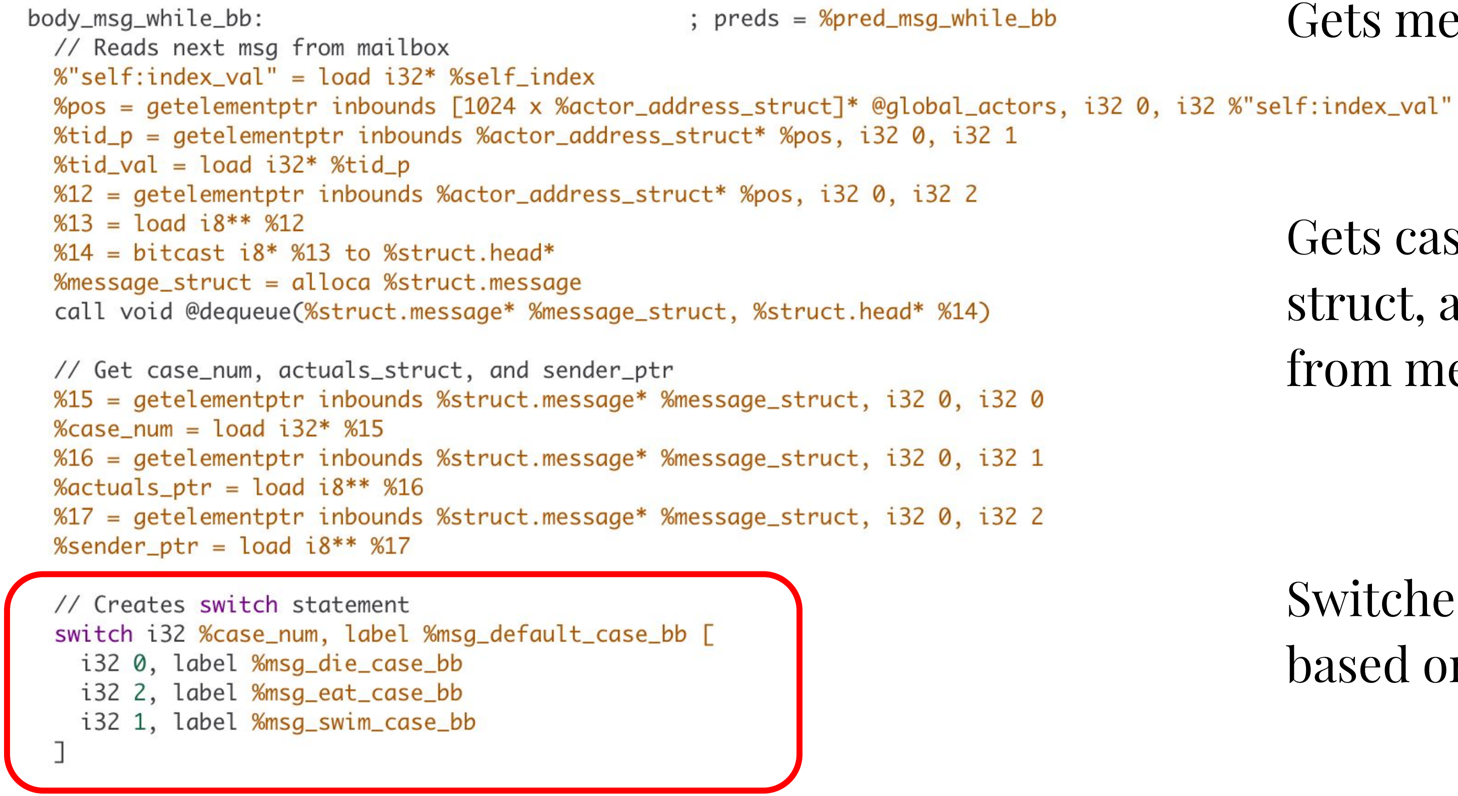


```
body_msg_while_bb:
```



```
// Exits while-loop and actor "dies"
merge_msg_while_bb:
%pred_msg_while_bb
  %ret = alloca i8
  ret i8* %ret
}
```

Implementation - Message Cases



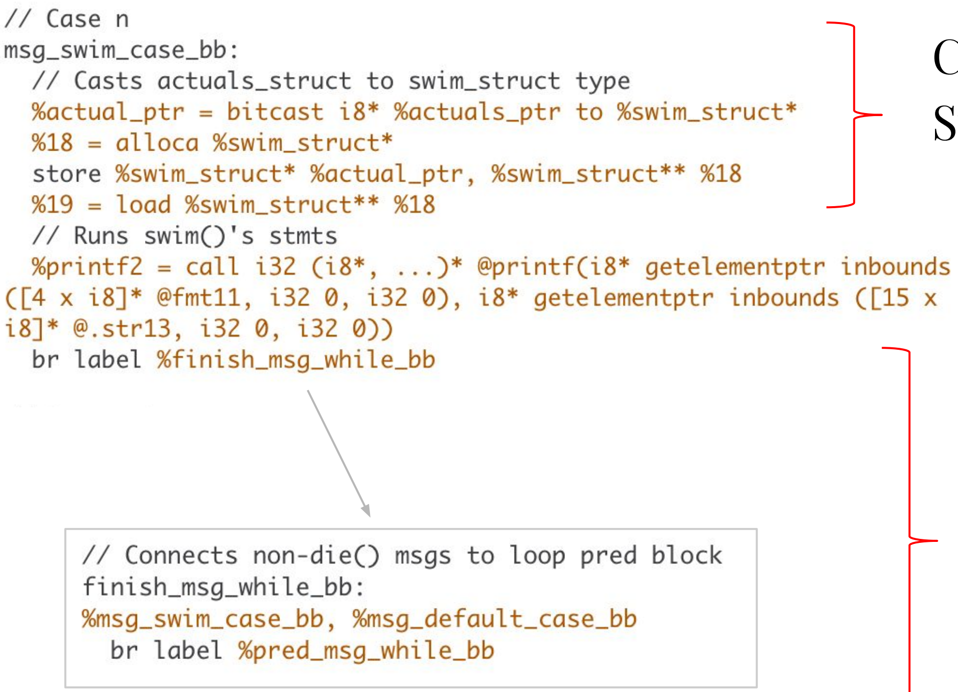
```
body_msg_while_bb:                                 ; preds = %pred_msg_while_bb
  // Reads next msg from mailbox
  %"self:index_val" = load i32* %self_index
  %pos = getelementptr inbounds [1024 x %actor_address_struct]* @global_actors, i32 0, i32 %"self:index_val"
  %tid_p = getelementptr inbounds %actor_address_struct* %pos, i32 0, i32 1
  %tid_val = load i32* %tid_p
  %12 = getelementptr inbounds %actor_address_struct* %pos, i32 0, i32 2
  %13 = load i8** %12
  %14 = bitcast i8* %13 to %struct.head*
  %message_struct = alloca %struct.message
  call void @dequeue(%struct.message* %message_struct, %struct.head* %14)

  // Get case_num, actuals_struct, and sender_ptr
  %15 = getelementptr inbounds %struct.message* %message_struct, i32 0, i32 0
  %case_num = load i32* %15
  %16 = getelementptr inbounds %struct.message* %message_struct, i32 0, i32 1
  %actuals_ptr = load i8** %16
  %17 = getelementptr inbounds %struct.message* %message_struct, i32 0, i32 2
  %sender_ptr = load i8** %17

  // Creates switch statement
  switch i32 %case_num, label %msg_default_case_bb [
    i32 0, label %msg_die_case_bb
    i32 2, label %msg_eat_case_bb
    i32 1, label %msg_swim_case_bb
  ]
```

Gets message.

Gets case num, actuals struct, and sender ptr from messages.

Switches to branch based on case num.

Implementation - For every message case,



```
// Case n
msg_swim_case_bb:
  // Casts actuals_struct to swim_struct type
  %actual_ptr = bitcast i8* %actuals_ptr to %swim_struct*
  %18 = alloca %swim_struct*
  store %swim_struct* %actual_ptr, %swim_struct** %18
  %19 = load %swim_struct** %18
  // Runs swim()'s stmts
  %printf2 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
([4 x i8]* @fmt11, i32 0, i32 0), i8* getelementptr inbounds ([15 x
i8]* @.str13, i32 0, i32 0))
  br label %finish_msg_while_bb
```

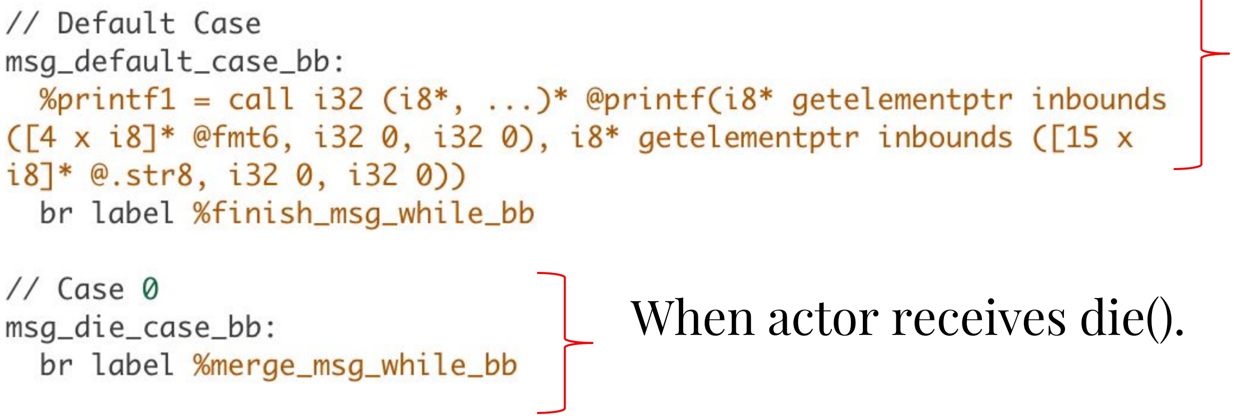
```
// Connects non-die() msgs to loop pred block
finish_msg_while_bb:
%msg_swim_case_bb, %msg_default_case_bb
  br label %pred_msg_while_bb
```

Casts Actuals Struct to Formals Struct.

Executes Message Stmts.

Branches back to while loop.

Implementation - Special Message Cases,



```
// Default Case
msg_default_case_bb:
  %printf1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
([4 x i8]* @fmt6, i32 0, i32 0), i8* getelementptr inbounds ([15 x
i8]* @.str8, i32 0, i32 0))
  br label %finish_msg_while_bb

// Case 0
msg_die_case_bb:
  br label %merge_msg_while_bb
```

When actor receives an unknown message.

Executes drop() code.

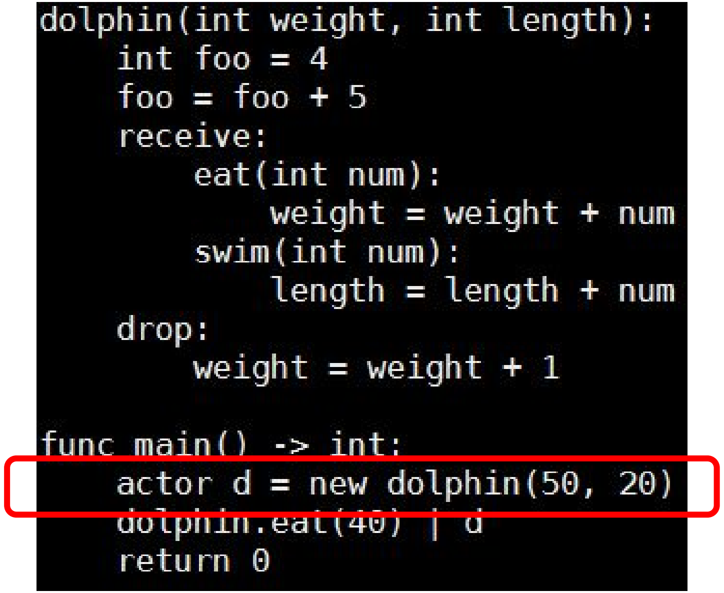
When actor receives die().

Implementation - Actors in Threads

Q: What happens when a new actor is created?

- A new message queue is created, and is passed along with formals as arguments to a `pthread_create` call running that actor type’s function

- Specifically: a struct is created containing the message queue pointer and the actuals, and a pointer to that is passed along with the function pointer to `pthread_create`

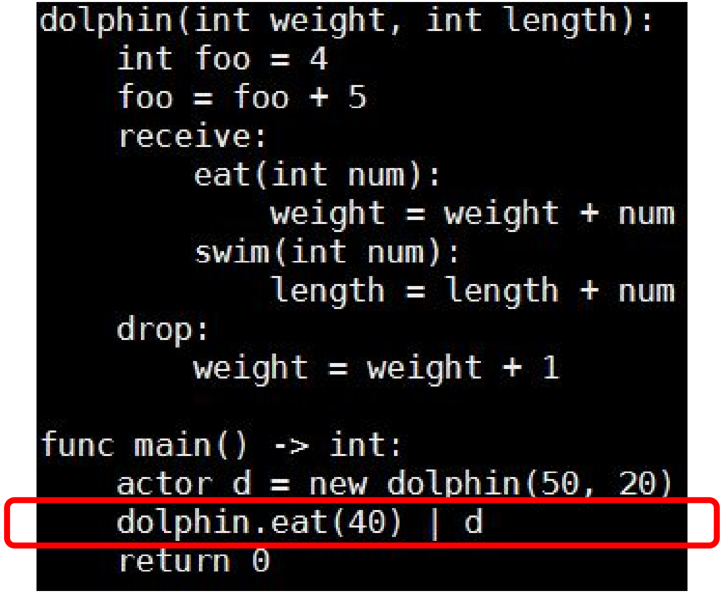


```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1

func main() -> int:
    actor d = new dolphin(50, 20)
    dolphin.eat(40) | d
    return 0
```

Implementation - Sending Messages to Actors

Q: How are messages sent to actors?

- `d` is resolved to a pointer to a message queue
- `dolphin.eat` is resolved to an int representing the case number in the actor’s switch statement at compile time
- A message struct is formed placing the case number and a struct containing the arguments and enqueued on `d`’s message queue

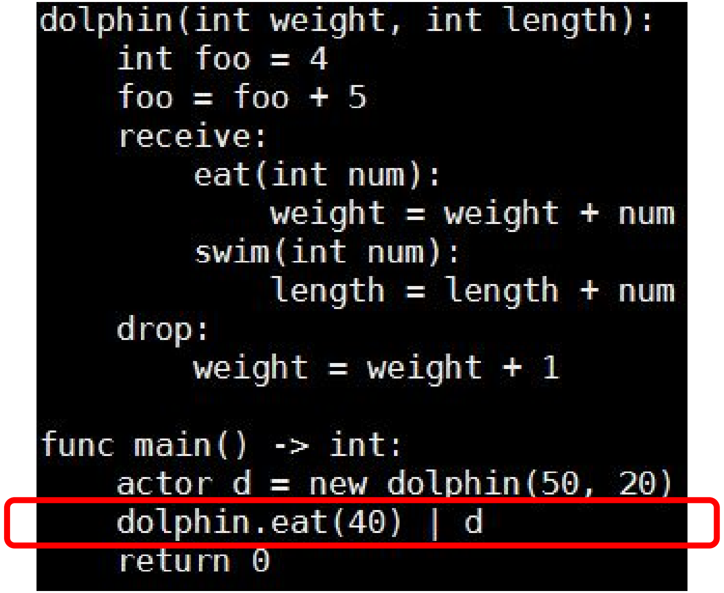


```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1

func main() -> int:
    actor d = new dolphin(50, 20)
    dolphin.eat(40) | d
    return 0
```

Implementation - Sending Messages to Actors

Q: How are messages sent to actors?

- The address of an actor resolves to its message queue!
- `d` can be passed around to other actors
- Anyone with the address of `d` can send it a message

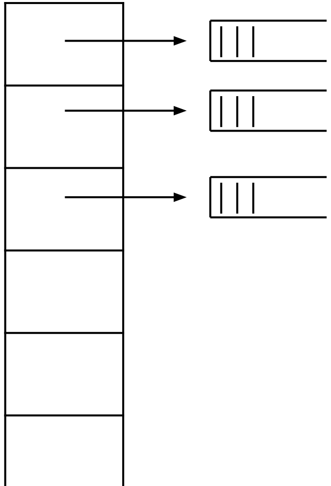


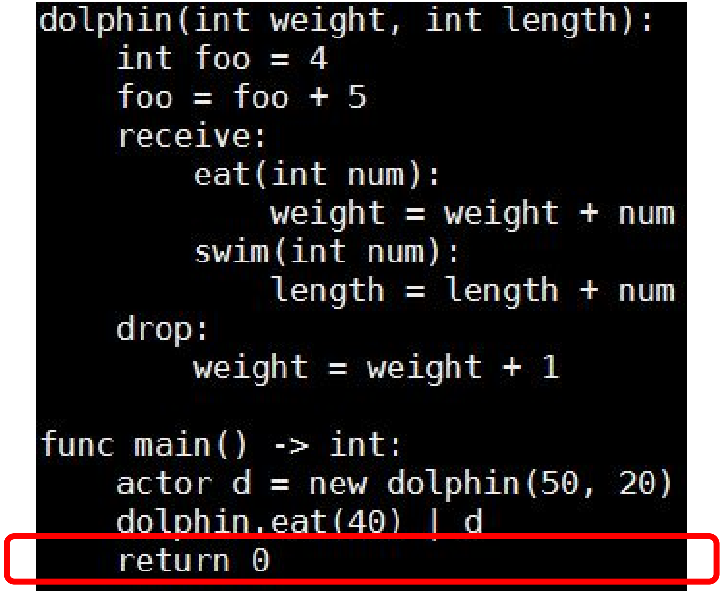
```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1

func main() -> int:
    actor d = new dolphin(50, 20)
    dolphin.eat(40) | d
    return 0
```

Implementation - Joining Actors and Metadata

Q: How are the threads joined?

- A global array of message queues is kept from the inception of the program
- When `main()` returns, it iterates over the array, joining each tid
- Metadata is also kept with the message queues (like tid)





```
dolphin(int weight, int length):
    int foo = 4
    foo = foo + 5
    receive:
        eat(int num):
            weight = weight + num
        swim(int num):
            length = length + num
    drop:
        weight = weight + 1

func main() -> int:
    actor d = new dolphin(50, 20)
    dolphin.eat(40) | d
    return 0
```

Demo