Named Optional Parameters - JavaScript Style

Brian Davidson (briandavidson@meta.com)

Summary

C++ developers often run into the situation where we have multiple parameters in a function with the same type. It is very easy to make accidental switcheroos (e.g. source/destination being flipped) without a compiler error.

```
void send(Address source, Address destrination{};
void send(Address destination{};
```

All below compile without warnings or readability hints

```
send(to,from); // flipped - should be from,to
send(from);    // meant to use to
```

Is there a way to make C++ function calls clear through naming, similar to python or javascript?

```
def send(source=None, destination=None):
send(source=from, destination=to)
send(source=from)
send(destination=to)
send(destination=from) # valid python - can C++ do better?
```

Optional named arguments are previously thought to not be possible in C++.
Is it possible to write such function calls in C++?

The benefits would be:
- Each argument is named, so it is clear what type is being used in what context
- Combinations of arguments can be represented polymorphically in overloaded function arguments forming a clear function call

Inspiration: Javascript

The anonymous object parameters of javascript provide a readable function API.

```
logger({one:0.9999});
> 0.9999
logger({one:"one",two:"two"});”
> "one" "two"
logger({one:1,two:2, three: "three"});
> 1 2 "three"
```

The limitation of javascript duck typing is that you only get one anonymous object removing the ability to overload, since the interpreter cannot distinguish by type.

```
function logger({one, two, three} = {}) {
// in signature: = {} provides destructure
// similar to structured binding

// first check for what would be a “compiler” error
   if ( one == null )
     throw new Error("i wish I was compiled!")
   else if ( one != null && two == null && three != null )
     throw new Error("i wish I was compiled!!")
  // dynamic dispatch
  else if ( three != null )
       console.log(one, two, three)
   else if ( two != null )
       console.log(one,two)
   else
       console.log(one)
```

C++ Implementation

The C++ implementation will create expressive API calls through utilizing combination of :
- POD structs -- structs with data members only
- designated initializers -- initializing a struct with: `{ .field = value }`
- structured binding -- disaggregating a struct with: `auto& [field1, field1] = myStruct`
- function overloading - defining the same function id multiple times, but with different typed arguments

The example function is processCarStateTransition that builds a path of 1...4 elements :
1. A vehicle type
2. A starting road state composed of a road type and speed
3. A transition from one road state to another, such as an intersection
4. Another road state.

The goal of this example is to build an API where the two road state transitions are explicitly stated and are statically dispatched to the correct implementation.

Callee Perspective

This first example has no room for accidentally switching the road state -- There is only one.

```
 // the .= syntax is a designated initializer used to assign POD values
auto result = processCarStateTransition({
      .car = Car::FastSportsCar,
      .startingRoadState = {.type = RoadType::Highway,.speed = Speed::FAST}});
```

This second example has room for error. The road transitions could be accidentally switched.

```
auto result = processCarStateTransition(
      {.car = Car::Jeep,
       .startingRoadState = {.type = RoadType::Trail,.speed = Speed::SLOW},
       .roadTransition = {Switches::Intersection, {false}},
       .nextRoadState = { .type = RoadType::Avenue,.speed = Speed::NORMAL}});
```

Ambiguity

The following structs will in theory dispatch to processCarStateTransition depending on how you call them.

```
struct PathUnchanged {
  Car car;
  RoadState startingRoadState;
};

struct PathToNextRoad {
  Car car;
  RoadState startingRoadState;
  Switches roadTransition;
  RoadState nextRoadState;
};
```

```
struct PathToChangingStatus {
  Car car;
  RoadState startingRoadState;
  Switches roadTransition;
};
```

```
// Overloaded functions that accept the structs
Answer processCarStateTransition(const PathUnchanged& path);
Answer processCarStateTransition(const PathToChangingStatus& path);
Answer processCarStateTransition(const PathToNextRoad& path);

// Compiles
  auto result = processCarStateTransition(PathUnchanged{
      .car = Car::FastSportsCar,
      .startingRoadState = {.type = RoadType::Highway, .speed = Speed::FAST},
  });

// Ambiguous - fails to compile
  auto result = processCarStateTransition({
      .car = Car::FastSportsCar,
      .startingRoadState = {.type = RoadType::Highway, .speed = Speed::FAST},
  });
```

Non-Ambiguous

Why was there ambiguity?
The RoadState type had
1. A default value when unspecified
2. Implicit POD construction could not differ between the three structs
   `{.type = RoadType::Highway, .speed = Speed::FAST}`
   Could be any of `PathUnchanged/ ..ToChangingStatus/ ..ToNextRoad`

The fix is to remove the ambiguity by removing the ability to implicitly construct without stating a value.

The Value struct illustrates what the 2...n types would need as a data member.

```
struct Value {
  Value() = delete;
  Value(bool state) : state_{state} {}

 private:
  bool state_;
};

struct RoadState {
  RoadType type;
  Value noDefaultExists;
  Speed speed;
};
```

Inside of processCarStateTransition the struct argument is spread with structured binding.

```
Answer processCarStateTransition(const PathToChangingStatus& path) {
   auto& [car, roadType, roadTransition] = path;
  return Answer::TWO;
}
```

First Class Feature Proposal

The main drawback of this approach are the hoops needed to jump through for removing ambiguity. Additionally, creating a struct just for arguments seems wasteful, despite the compiler potentially optimizing it away.

This can be improved upon with some potential language tweeks that makes the struct syntactic sugar

Option 1: Explicit Parameter List as Struct

Here the curly braces imply an anonymous struct just like JavaScript.

```
Answer processCarStateTransition({Car car, RoadState
startingRoadState, Switched roadTransition, RoadState
finalRoadState}) {
  //... variabled local for free “struct”is syntactic sugar
}
```

Option 2: Implicit Parameter List as Struct

No curly braces are needed. All functions have an implied anonymous struct.

Option 3: Both

Curly braces at parameter declaration requires function calls to use struct.
Otherwise, not having curly braces allows for implied struct.