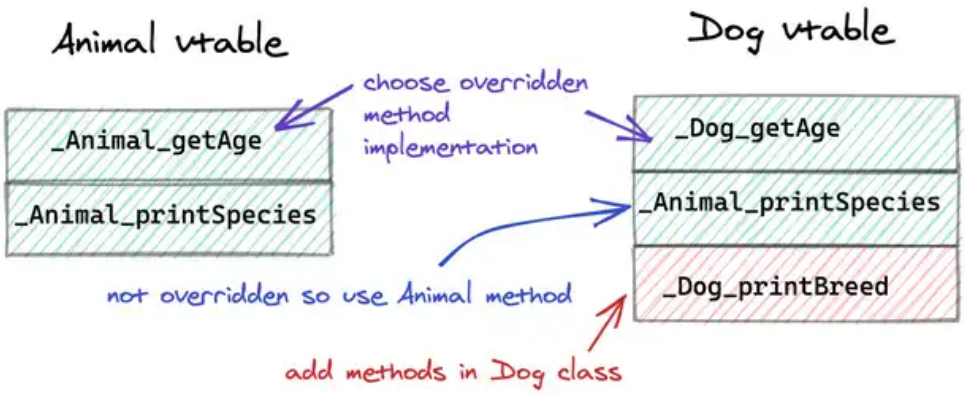
CREATING THE BOLT COMPILER: PART 11

Adding Inheritance and Method Overriding to Our Language

JANUARY 25, 2021 7 MIN READ



SERIES: CREATING THE BOLT COMPILER

Part 1: How I wrote my own "proper" programming language
Part 2: So how do you structure a compiler project?
Part 3: Writing a Lexer and Parser using OCamllex and Menhir
Part 4: An accessible introduction to type theory and implementing a type-checker
Part 5: A tutorial on liveness and alias dataflow analysis
Part 6: Desugaring - taking our high-level language and simplifying it!
Part 7: A Protobuf tutorial for OCaml and C++
Part 8: A Complete Guide to LLVM for Programming Language Creators
Part 9: Implementing Concurrency and our Runtime Library
Part 10: Generics - adding polymorphism to Bolt
Part 11: Adding Inheritance and Method Overriding to Our Language

TABLE OF CONTENTS

Welcome back to part 11 of the series! We've got concurrency and generics in our language, but we're still missing a key component: **inheritance**. Remember, the four main OOP principles are

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Ah yes, there's polymorphism to tackle too. We've seen *ad-hoc* polymorphism, when we implemented method overloading in the desugaring stage. We've seen *parameteric* polymorphism, when we implemented generics last time round. In this post we'll cover the third type of polymorphism: subtype polymorphism - **method overriding**.

By the end of the tutorial, we'll be able to compile the following program. We'll refer to this example to *motivate* the changes needed to the compiler, so that not only will you understand how the changes work, but you'll understand *why* they are implemented in that way.

And hopefully, you'll be able to see the general principles we've used to add generics in the last post, and inheritance and method overriding in this post, so you'll be able to add even more language features going forward!

Copy

```
class Breed {...}
class Species { ... }
class Animal {
  int age;
  Species species;
  int getAge() {
    return this.age
  }
  void printSpecies(){ ... }
}
class Dog extends Animal {
  Breed breed;
  int getAge() {
    return 7*this.age // dog years!
  }
  void printBreed(){ ... }
}
```

```
function void printAge(Animal a){
  printf("I'm %d years old!", a.getAge());
}
void main() {
  let animal = new Animal(age: 2);
  let dog = new Dog(age: 2);

  printAge(animal) // print 2
  printAge(dog) // print 14
}
```

Just give me the code!

As with the rest of the series, all the code can be found in the Bolt repository.

If you want to see the specific commits that were needed to implement inheritance, check out this pull request and this pull request.

AST definitions

We need to store this inheritance relationship in our Abstract Syntax Tree for our compiler to access. The easiest way is to store the name of the superclass in the class definition, since we're reading it in when we parse `CLASS Someclass EXTENDS Otherclass`:

parsed_ast.mli

Copy

```
type class_defn = TClass of
      Class_name.t
      * generic_type option
      * Class_name.t option (* optional superclass *)
      * capability list
      * field_defn list
      * method_defn list
```

Type-Checker

When we're adding a new language feature, we need to determine what effect it has on the existing typing rules. With inheritance and method overriding we have the following new rules:

- Subclasses like `Dog` have access to not only their own fields and methods, but also their superclass `Animal`'s fields and methods, (and the fields and methods of their superclass' superclass, and so on up the inheritance hierarchy).
- Overridden methods (`getAge`) need to have the same type signature as their superclass (their return types need to match), since they're being used in the same contexts.
- If a superclass is generic, then the subclass must also be generic - otherwise how could it access a field of generic type `T` in the superclass?
- Subclasses like `Dog` are **subtypes** of their superclass (`Animal`). We have some new typing rules to handle when you can use `Dog` in place of `Animal`.

Accessing superclass' methods

We just need to update the `get_class_methods`, `get_class_fields`, methods etc. to recursively look up the method/field first in the current class, then its superclass and so on.

Here's a simple example: each class has a list of capabilities. With inheritance, we do a recursive check to get the superclass' capabilities too. The other `get_class_` methods are similar:

type_env.ml

Copy

```
let rec get_class_capabilities class_name class_defns =
  let open Result in
  get_class_defn class_name class_defns Lexing.dummy_pos
  >>= fun (Parsed_ast.TClass (_, _, maybe_superclass, capabilit
  ( match maybe_superclass with
  | Some superclass -> get_class_capabilities superclass class_
  | None            -> Ok [] )
  >>| fun superclass_caps -> List.concat [superclass_caps; capa
```

Generics and Inheritance

We pattern-match - if the current class isn't generic, and the superclass is generic, raise a type error!

type_inheritance.ml

Copy

```
let type_generics_inheritance class_name curr_class_maybe_gener
    (Parsed_ast.TClass (superclass_name, superclass_maybe_gener
  match (curr_class_maybe_generic, superclass_maybe_generic) wi
  | None, None | Some Generic, None | Some Generic, Some Generi
  | None, Some Generic ->
      Error
        (Error.of_string
           (Fmt.str
              "Type error: class %s must be generic since super
              (Class_name.to_string class_name)
              (Class_name.to_string superclass_name)))
```

Method overriding

Remember, a method *overrides* another if it has the same name and parameter types (as `getAge` does in our example). To check the overriding methods have the right type, we look up the method type signatures of the current class and the superclass' methods. We raise an error if the current class has a method that overrides an inherited method (same method name and parameter types) but differs in return type.

type_inheritance.ml

Copy

```
let type_method_overriding class_name class_defns method_defns
  let open Result in
  get_methods_type_sigs method_defns
  |> fun methods_type_sigs ->
  get_methods_type_sigs  (get_class_methods class_defns supercl
  |> fun inherited_methods_type_sigs ->
  List.filter
    ~f:(fun (meth_name, ret_type, param_types) ->
      List.exists
        ~f:(fun (inherited_meth_name, inherited_ret_type, inher
```

```
            meth_name = inherited_meth_name
            && param_types = inherited_param_types
            && not (ret_type = inherited_ret_type))
          inherited_methods_type_sigs)
    methods_type_sigs
  |> function
  (* we find any methods that violate overriding types *)
  | []                      -> Ok ()  (* no violating methods *)
  | (meth_name, _, _) :: _ ->
      Error ...
```

Subtyping

We say a type `A` *subtypes* type `B`, if we can use `A` in place of `B`. That occurs if they're equal (obvious) or if `A` is a subclass of `B` e.g. `Dog` in place of `Animal`. Equivalently, we can say B is the *supertype* of A.

type_inheritance.ml

Copy

```
let is_subtype_of class_defns type_1 type_2 =
  type_1 = type_2
  ||
  match (type_1, type_2) with
  | TEClass (class_1, type_param_1), TEClass (class_2, type_par
      type_param_1 = type_param_2 && is_subclass_of class_defns
  | _ -> false
```

Intuitively the subtype, `A`, has more info than `B`: the `Dog` class has all the `Animal` behaviour and then some more behaviour specific to dogs.

When do we use subtyping?

Subtyping in Variable Assignments

We can assign subtypes to variables e.g. `let x: Animal = new Dog()`. So in our `let` expression typing judgement, we ensure the bound expr is a subtype of the type annotation:

type_inheritance.ml

Copy

```
| Parsed_ast.Let (loc, maybe_type_annot, var_name, bound_expr)
      ...
      type_with_defns bound_expr env
      >>= fun (typed_bound_expr, bound_expr_type) ->
        ( match maybe_type_annot with
      | Some type_annot ->
          if is_subtype_of class_defns bound_expr_type type_ann
      ...
```

Subtyping in Functions

Subtyping rules for functions are a little more complicated, so we'll use our intuitive notion of subtypes having more info to help us out here.

We can pass subtypes as arguments, as if the function `printAge` was expecting an `Animal` and we give it a `Dog`, it can ignore the extra info like `breed`. Here's the snippet of code that does the check:

type_overloading.ml

Copy

```
...
if are_subtypes_of class_defns args_types param_types then
        Ok (param_types, return_type)
...
```

> Hold on, you ask, why is this subtype check in the `type_overloading` file?
>
> Well, as you add language features, they start to interact with each other. Earlier in the series, we added function overloading: i.e. multiple functions with the same name but different parameter types. We choose the correct overloaded function based on the argument types: we pick the function whose parameter types *match*. Before, our definition of *match* was that the types were equal. Now, we say they *match* if the

> argument types are **subtypes** of the param types. All the details are in the code.

Likewise, when type-checking our function return type, the body type *matches* the function return type, if it is a subtype. (If the function returns `void` we don't care about the body type.)

type_functions.ml

Copy

```
...
>>= fun (typed_body_expr, body_return_type) ->
 if return_type = TEVoid || is_subtype_of class_defns body_retu
    Ok ...
else Error ...
```

An identical check is done to type-check methods.

Lowering to LLVM

Now we've done the type-checking, we need to output LLVM IR in order to run the program. Let's quickly remind ourselves of the program we're trying to compile:

Copy

```
class Animal {
  int age;
  Species species;
  int getAge() {
    return this.age
  }
  void printSpecies(){ ... }
}
class Dog extends Animal {
  Breed breed;
  int getAge() {
    return 7*this.age // dog years!
  }
  void printBreed(){ ... }
```

```
}
function void printAge(Animal a){
  printf("I'm %d years old!", a.getAge());
}
void main() {
  let animal = new Animal(age: 2);
  let dog = new Dog(age: 2);

  printAge(animal) // print 2
  printAge(dog) // print 14
}
```

Inheritance and Structs

Both the `Animal` and `Dog` classes are desugared to structs containing their fields. This desugaring is straightforward for `Animal`:

Copy

```
struct Animal {
  int age;
  Species *species;
}
```

Accessing the `age` field is desugared into getting a pointer to field `0` of the struct, and likewise `species` is field `1`.

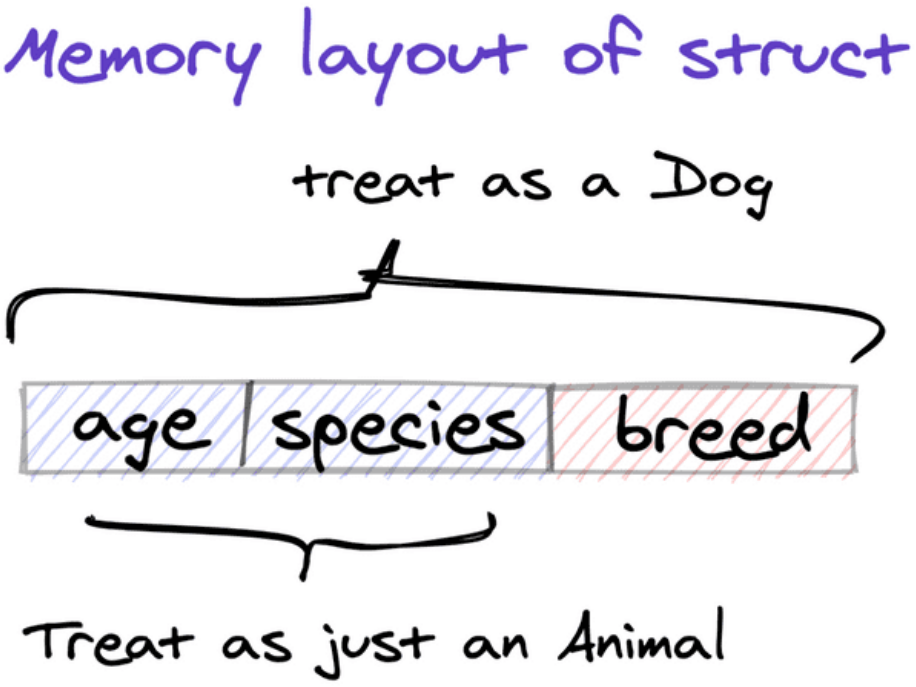
On to the struct for `Dog`. We have three requirements:

- The struct needs to contain the fields in the `Dog` class.
- It also needs to contain the fields in the `Animal` class.
- A `Dog` struct should be able to be used wherever an `Animal` struct is expected.

The last point is the critical one. Say I have a `dog` object being treated as an `Animal`. When I query `dog.age` and `dog.species`, I'll expect them to be in field `0` and `1` of the struct, since it's of type `Animal`. So the `Dog` struct needs to preserve that field indexing. So the only place the field `breed` can go is at index `2`.

Copy

```
struct Dog {
  int age;
  Species *species;
  Breed *breed;
}
```



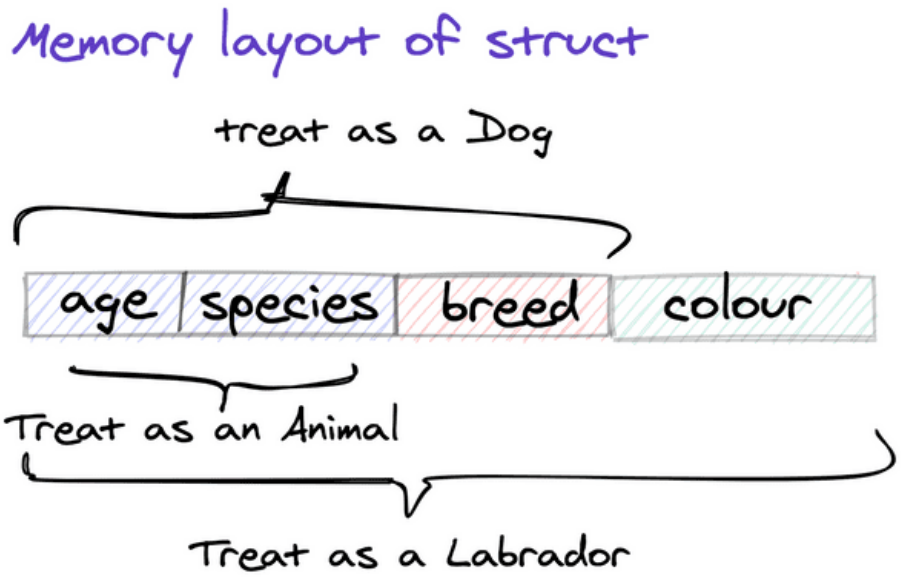
In general, we order the struct so all the superclass's fields go first, *then* any fields declared in the current class. If we added a subclass `Labrador`:

Copy

```
class Labrador extends Dog{
  Colour colour;
}

// Desugared
struct Labrador {
  int age;
  Species *species;
  Breed *breed;
  Colour *colour;
```

```
}
```



If we want to treat a `Labrador` as an `Animal`, only look at the first 2 fields. If you want to treat it as a `Dog` look at the first 3 fields. And so on.

You can see this ordering in our `get_class_fields` method during our IR generation stage, which puts `superclass_fields` first.

ir_gen_env.ml

Copy

```
let rec get_class_fields class_name class_defns =
  get_class_defn class_name class_defns
  |> fun (TClass (_, maybe_superclass, _, fields, _)) ->
  ( match maybe_superclass with
  | Some super_class -> get_class_fields super_class class_defn
  | None             -> [] )
  |> fun superclass_fields -> List.concat [superclass_fields; f
```

Since we've handled this in the Bolt IR gen stage of the compiler frontend, we don't need to change our LLVM backend, right?

Almost right. Just one hitch: LLVM IR is typed, so it will complain if we pass a `Dog *` pointer to a function that expects a `Animal *` pointer. Remember, LLVM has no notion of inheritance, just raw structs, so it can't see that `Dog` is a subclass of `Animal`. We thus *explicitly cast* the argument pointer to the function's expected param type before we pass it to the function:

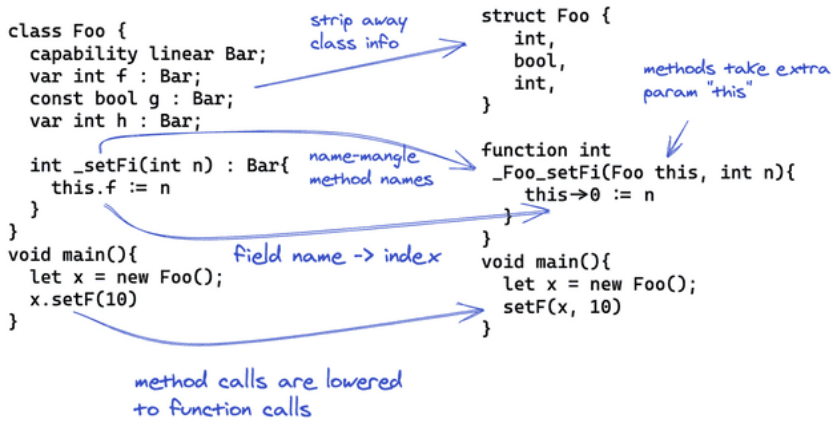
expr_codegen.cc

Copy

```
std::vector<Value *> argVals;
  for (int i = 0; i < expr.arguments.size(); i++) {
    Value *argVal = expr.arguments[i]->codegen(*this);
    Type *paramTy = calleeFunTy->getParamType(i);
    Value *bitCastArgVal = builder->CreateBitCast(argVal, param
    argVals.push_back(bitCastArgVal);
  }
```

Method Overriding and Virtual Tables

Method overriding is a bit tricky, so take a breather. From our previous post, we have that methods are desugared to regular functions:



The diagram from our previous desugaring post.

In our running example in this post we have `getAge` overridden, so we have these two functions corresponding to the implementation of `getAge` in each of the classes:

Copy

```
Animal_getAge(Animal this){
  return this.age
}

Dog_getAge(Dog this){
  return 7 * this.age
```

```
}
```

When we call `a.age()` in our `printAge` function, we want to call `Animal_getAge` if the underlying object is an `Animal`, and `Dog_getAge` if the object is actually a `Dog`. The thing is, in general we don't know this information at compile-time. Here's an example where type of `dog` is only known at runtime:

Copy

```
let dog = new Animal()
if (someCondition) {
  dog = new Dog()
}
dog.getAge() // which function do we call?
```

So how do we insert the right call? The problem is not too dissimilar to this:

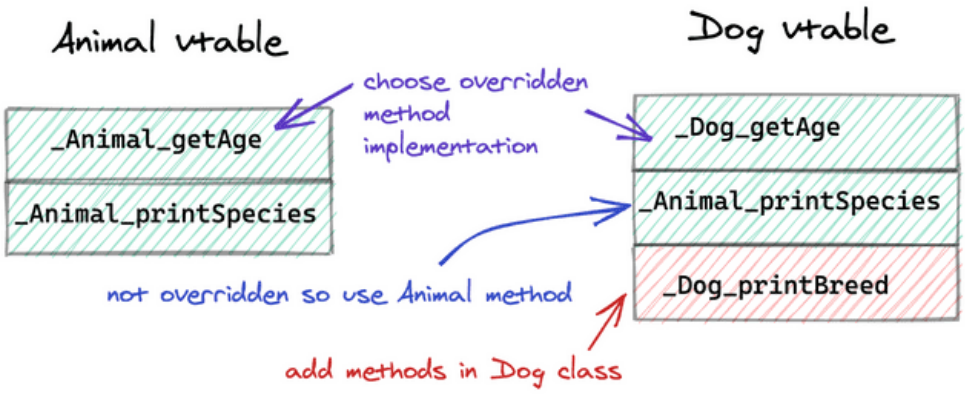
Copy

```
let dog = new Animal()
dog.age = 7
if (someCondition) {
  dog.age = 14
}
dog.age // which value does age have?
```

We know how to do this: look up the value by indexing into the *list of fields* in the `dog` struct at runtime (index `0` for `age`).

Since we've solved the problem for fields, let's do the same thing for methods. We can create a *list of (pointers to) methods* for each class, called the **virtual table** or **vtable**. Instead of determining the function call at compile-time, we instead provide an *index* into the table and call the function pointed to at that index at runtime.

For similar reasons to the struct fields before, superclasses' methods come before the current class's methods in the vtable. So we have `getAge`, `printSpecies` first as they're from `Animal`, then we have `getBreed` from `Dog`. To override a method, we just replace its entry in the table:



Now we can just say for `dog.getAge()`, look at index `0` of the `dog`'s vtable, and execute whichever function's there. Or index `1` for `dog.printSpecies()`. **Problem solved!**

Virtual Tables in our structs

Unlike fields, which are different for each object, vtables are the same for *all* objects of a *given class*. Having a copy of the same vtable in each object is wasteful: instead have just **one** global vtable **per class**, and have the objects store a *pointer* to that.

We've reserved the first field in our object struct for the vtable pointer:

Copy

```
struct Animal {
  AnimalVTable *vtableptr;
  int age;
  Species *species;
}

struct Dog {
  DogVTable *vtableptr;
  int age;
  Species *species;
  Breed *breed;
}
```

Okay, let's implement it with some code! To generate the vtable, we get a list of the class' methods, **annotated** with the class they came from (so we get the right overridden method). That is, we don't have `getAge` we have the pair `(Dog,`

`getAge)` . We use this pair to generate the name-mangled function name: `_Dog_getAge` . Our vtable is just a list of these name-mangled function names.

ir_gen_env.ml

Copy

```
let ir_gen_vtable class_name class_defns =
  get_class_annotated_methods class_name class_defns
  |> fun class_annotated_methods ->
  List.map
    ~f:(fun (class_annot, meth_name) -> name_mangle_method_name
    class_annotated_methods
```

LLVM implementation of Virtual Tables

We implement VTables as a struct of function pointers. Below are the type definitions, and the corresponding global variable declaration.

vtable.ll

Copy

```
%_VtableAnimal = type { i32 (%Animal*)*, void (%Animal*)* }
%_VtableDog = type { i32 (%Dog*)*, void (%Animal*)*, void (%Dog
}

@_VtableAnimal = global %_VtableAnimal { i32 (%Animal*)* @_Anim
@_VtableDog = global %_VtableDog { i32 (%Dog*)* @_Dog__getAge,

%Animal = type { %_VtableAnimal*, i8*, i32, i32, i32, %Species*
%Dog = type { %_VtableDog*, %Species*, %Breed* }
```

Creating the global vtables follows the format we mentioned for global variables in the LLVM post. We create the table as a `ConstantStruct` , and populate the (`vector<Constant *>`) body with `Function *` pointers to each of the methods.

class_codegen.cc

Copy

```
void IRCodegenVisitor::codegenVTables(
    const std::vector<std::unique_ptr<ClassIR>> &classes) {
  for (auto &currClass : classes) {
    std::string vTableName = "_Vtable" + currClass->className;
    StructType *vTableTy =
        module->getTypeByName(
          StringRef(vTableName));
    std::vector<
    Constant *> vTableMethods;
    std::vector<
    Type *> vTableMethodTys;
    for (auto &methodName : currClass->vtable) {
      Function *method = module->getFunction(
        StringRef(methodName));
      vTableMethods.push_back(method);
      vTableMethodTys.push_back(method->getType());
    }
    vTableTy->setBody(vTableMethodTys);
    module->getOrInsertGlobal(vTableName, vTableTy);

    GlobalVariable *vTable = module->getNamedGlobal(vTableName)
    vTable->setInitializer(
      ConstantStruct::get(vTableTy, vTableMethods));
  }
}
```

Then to call the function, we're replacing our static function call:

Copy

```
*calleeMethod =
      module->getFunction(expr.methodName);
```

with a vtable lookup. This is just two Struct GEP lookups: first we get the vtable pointer by reading index `0` of the object, then we get the method pointer by using the `methodIndex` into the vtable. (Again, check the LLVM post for a refresher on LLVM).

expr_codegen.cc

Copy

```
Value *vTablePtr = builder->CreateLoad(builder->CreateStructGEP
    thisObj->getType()
        ->getPointerElementType() /* get type of element on hea
    thisObj, 0));
Value *calleeMethodPtr =
    builder->CreateStructGEP(vTablePtr->getType()->getPointerEl
                             vTablePtr, expr.methodIndex);

Value *calleeMethod = builder->CreateLoad(calleeMethodPtr);
```

> An minor detail, why is `calleeMethod` now of type `Value *` not `Function *` ? I asked the LLVM dev mailing list this question - it's because `Function *` refers to a function known at compile-time. Our vtable functions are looked up at runtime, so aren't of type `Function *` .

Summary

This post on inheritance and method overriding brings to a close the Bolt compiler series for now. These tutorials cover the general language features of the Bolt language at the time I submitted my dissertation.

There's always more language features to add, perhaps later down the line I'll write another tutorial on arrays! In the meantime, I've got a host of new posts on other content coming this year - 40 new posts coming in 2021!

Share This On Twitter

If you liked this post, please consider sharing it with your network. If you have any questions, tweet away and I'll answer :) I also tweet when new posts drop!

> **PS:** I also share helpful tips and links as I'm learning - so you get them **well before** they make their way into a post!
>
> Post Follow @mukulrathi_

SERIES: CREATING THE BOLT COMPILER

Part 1: How I wrote my own "proper" programming language
Part 2: So how do you structure a compiler project?
Part 3: Writing a Lexer and Parser using OCamllex and Menhir
Part 4: An accessible introduction to type theory and implementing a type-checker
Part 5: A tutorial on liveness and alias dataflow analysis
Part 6: Desugaring - taking our high-level language and simplifying it!
Part 7: A Protobuf tutorial for OCaml and C++
Part 8: A Complete Guide to LLVM for Programming Language Creators
Part 9: Implementing Concurrency and our Runtime Library
Part 10: Generics - adding polymorphism to Bolt
Part 11: Adding Inheritance and Method Overriding to Our Language

← Generics - adding polymorphism to Bolt

I've Started a YouTube Channel →