Lecture 18: Handling Object Oriented Language Features during Compilation

By Dr. Muhammad Nur Yanhaona

From our knowledge of computer architecture, we can confidently say that modern computers are capable of arithmetic/logic operations, single-dimensional array access-like load/store from and to the memory, and jumping to a different instruction in the binary code conditionally/unconditionally. That is all there is in terms of the capability of machines. Using these features, conventional structural languages features are translated to machine code sequences as follows:

1. Branching logic such as if-else blocks, and switch-cases are implemented using conditional and unconditional jumps.
2. While and for loops are translated as code blocks just like branching code blocks with the exception that there is an unconditional jumping back to the beginning after the last instruction of the block.
3. Function calls are also like jumping to a different location of the binary code where the instructions corresponding the function are stored. Here the difference is that, the compiler, inserts code to push the local variables, return address of the caller of the function in a stack segment of the program memory along with the function call parameters. When the function returns back to the caller using a special type of unconditional jump, the variables are restored from the stack.
4. Multidimensional array accesses are translated into linear array index accesses using the array type information. Then simple load/store instructions are used.

However, if we look into modern-day object-oriented programming languages, we see a lot of interesting features, such as, user defined types/classes, class member functions, interfaces and abstract classes, field and method inheritance, method polymorphism, public/private/protected visibility control. How do these features get translated in machine code then? There are some standard techniques to process OOP features in compilers. Here we will investigate just one to resolve the mystery of OOP.

Fields of User Defined Types

Note that user defined types were already available in structural programming languages such as Cobol and C under various names such as structure or record. Actually, without type-specific methods, that are supported in user defined classes of Java/C++/C#, custom types with specific properties are a mere variable grouping feature. They get translated into single-dimensional array accesses in the generated binary code. Let us understand the compilation techniques with the below example of a C structure.

```
struct Student {
        int studentId;
        float cgpa;
        int batch;
        char[20] fullName;
}
```

When a typical C compiler sees a structure definition like the above, it creates a symbol table for the structure. We know a symbol table retains information about the type, width, and offset of its variable entries. Consequently, the symbol table for the above structure may look like as follows.

Name	Type	Width	Offset
studentId	Int	4	0
ggpa	Float	8	4
batch	Int	4	12
fullName	array(20, char)	40	16

Later, suppose you created a student object inside your code and assigned it to a variable name 'adam' and have the following statement in your program.

```
adam.cgpa = 3.85;
```

Assuming, the variable 'adam' has been allocated to @adam offset in your program's logical address space. Then the three-address code for the above line would be as follows:

```
@adam[4] = 3.85
```

What will happen to the following statement?

```
ch = adam.fullName[10];
```

A simple way to represent the above in the three-address code is as follows:

```
t1 = @adam[16]
ch = t1[20]
```

From this above example, we should have an idea how a compiler translates property accesses when a user defined type has another user defined type property inside it. For example, assume a Course structure has an instructor property, which is a Teacher type structure as follows:

```
struct Teacher {
        int employeeId;
        char[3] initials;
        char[20] name;
        char[15] position;
}
```

```
struct Course {
        int code;
        char[20] name;
        Teacher instructor;
}
```

Then suppose you have a cse420 course object in your program and trying to access the initials of the instructor of that course as follows:

```
initials = cs420.instructor.initials;
```

Then the compiler can translate the above statement as follows (assuming @cse420 is memory location of the course object.):

```
t1 = @cse420[44];
initials = t1[4];
```

That is, the compiler will use the symbol table of the Course structure to determine how to access the instructor property. Then it will use the symbol table of the Teacher structure to determine how to access the initials property.

Public/Private/Package/Protected Visibility Control

Well, there is no such thing as visibility control of variables and methods in the generated binary executable. Visibility control is a strictly design feature of OOP languages that a compiler peels off from the source code after semantic analysis. This means, the OOP compiler checks, during semantic analysis, if the fields and methods are accessible from different statements of your code according to the public, private, package, and protected modifiers you put in different class definitions of your program. If there is any violation in any statements, for example, you are trying to access a private property of Class B from Class A, then you get a semantic error. However, after the semantic analysis is done and the validity of access is ensured, these visibility control features are completely ignored by the compiler as it goes on generating the executable pretending those instructions never existed.

Among other things, an OOP compiler may also ignore Final and Constant keywords beside visibility control instructions. Similarly, that you cannot create an instance of an abstract class is also a validation done during semantic analysis and is ignored during later code generation phases.

Class Member Functions

One central aspect of OOP languages is that their classes have member functions that define behaviors of the objects the class represents. For example, suppose we have a Student Class having a list holding the GPAs of a student in different semesters. We can have a computeCGPA method that will sum up the GPAs from the list and return an average as the CGPA of the student. The question is how this is achieved, when at the binary code level, the only type of functions we have are regular functions where code can simply jump to then eventually return form. There are couple of clever tricks working in here. We will check them one by one. First let us define the following class definition as the reference example for our discussion.

```
Public Class Student {
        Private int id;
        Private String name;
        Private List<float> semesterGPAs;
        Public Boolean isInProbation(float thresholdCGPA) {
                float gpaSum = 0.0;
                foreach (semGPA in semesterGPAs) {
                        gpaSum += semGPA;
                }
                float cpga = gpaSum / semesterGPA.length();
                if (cgpa > thresholdCGPA) {
                        return false;
                }
                return true;
        }
}
```

By now we know that public/private etc. modifiers are ignored after semantic validation and there is a symbol table associated with each class holding information about its fields (aka. properties). So let us focus on the 'isInProbation' method only. During the semantic analysis of the method, the compiler would generate a symbol table for the method scope also and will determine the scope of the various variables used in it. From that, it would know that:

1. thresholdCGPA is in the function parameter scope.
2. gpaSum, semGPA, and cgpa variables are in the function body, that is, local scope.
3. semesterCGPAs is in the class scope.

After this analysis, the compiler will replace the class method with a generic global function as we find structural programming languages as follows:

```
Public Boolean isInProbation(Student this, float thresholdCGPA) {
        float gpaSum = 0.0;
        foreach (semGPA in this.semesterGPAs) {
                gpaSum += semGPA;
        }
        float cpga = gpaSum / semesterGPA.length();
        if (cgpa > thresholdCGPA) {
                return false;
        }
        return true;
}
```

Then everywhere in the source code the compiler finds an invocation to the member function from any student object as follows:

```
Boolean probation = studentA.isInProbation(threshold);
```

It would generate an alternative code in the intermediate representation as follows:

```
Boolean probation = isInProbation(studentA, threshold);
```

That does the trick of invoking a member function. Hence, at the end, invoking a member function of a class object is actually invoking a global function with the object as the first parameter. Isn't this clever?

Class Inheritance and Method Polymorphism

The most interesting aspect of dealing with OOP code in compiler is probably the handling of inheritance and polymorphism. Assume, we have the following base and derived classes in our program.

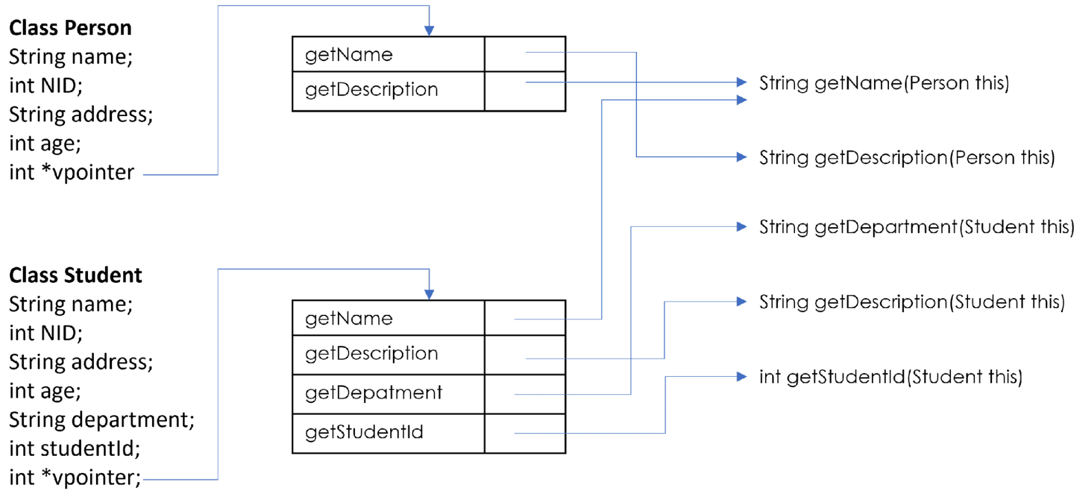
```
Class Person {                                  Class Student extend Person {
        String name;
        int NID;
        String address;
        int age;
        String getName() { return name; }
        String getDescription() {
                String desc = name
                        + ", " + address;
                Return desc;
        }
}
```

```
        String department;
        int studentId;
        String getDepartment() { return department; }
        String getDescription() {
                String desc = name + ","
                        + studentId + "," + department;
                Return desc;
        }
        Int getStudentId() { return studentId}
}
```

We know how member functions are translated by the compiler from the earlier discussion. From the above two classes, the compiler will generate the following five functions (I am going to ignore the body of the functions as you already know what changes are done there.)

1. String getName(Person this)
2. String getDescription(Person this)
3. String getDepartment(Student this)
4. String getDescription(Student this)
5. Int getStudentId(Student this)

Notice that there are two implementations of the getDescription function as the sub-class overrides the base class method. Then the compiler generates a table – called a vtable – for each class where the entries of the table refer to the specific implementation of the functions appropriate for that class's object. The compiler, finally appends a new pointer property in the classes – typically named vpointer – that refers to the table. So overall, the converted class descriptions look like as follows:



So overall, the steps of a member function access in OOP languages supporting type polymorphism are as follows:

1. Add the code for accessing the vpointer property of the class object.

2. Then call a generic global method that will take in the table vpointer refers to and the name of the function as input and return the location of the original member function to invoke.
3. Add the object as the first 'this' param to the function call.
4. Then add the remaining original parameters as specified in the program.
5. Then call the function at location received in Step 2.

You may wonder, why the compiler needs to generate vtables and vpointers and involve so many steps in the function calls. The reason is that, you can assign a sub-class object to a super-class type variable at runtime. Consequently, the compiler does not know the appropriate type of an object when it is generating the code. The type is only revealed during program execution. An indirection through the vpointer ensures that you are calling the right function despite that the apparent type may look different in the source code.