Object-Oriented Languages

CS152

Chris Pollett

Oct. 15, 2008.

Outline

- Object Orientation Motivated by ADT concerns
- Object Orientation as Message Passing.
- Just so stories of Preprocessor Implementations.
- Starting Objective-C

Introduction

- So far in this course, we have looked at the how to specify the syntax of a programming language, how to parse this syntax, and how the parsed result can be given a semantics -- at least at the level of declaration.
- We have been programming in C, a procedural language, although I am assuming in earlier courses you have been exposed to an object-oriented language such a Java and C++.
- Today, we are going to consider how language construct like object-orientation might evolve.
- We will give a "just-so story" of this evolution which probably has little connection to reality.
- The point though is to see how we might give a semantics for a new way of doing things in terms of a semantics which we already have defined.

ADTs

- Since Cobol there have been mechanisms to group together several related data items into one unit that can be manipulated. (**Encapsulation**)
- In C the mechanisms to do these kind of grouping are struct's and union's. Often you hear a struct kind of grouping called a **record** and the union kind, a **variant record**.
- Given such a record it is natural to try to group with it the functions associated with manipulating it.
- The combined structure is then an **abstract data type (ADT)**.
- Examples ADTs include PriorityQueues, Tables etc.
- Further, we often want to hide the implementation details from the user so that they can be improved as needed. (**Information hiding).**

More on ADTs

- To some degree, C allows one to separate interfaces from implementations using headers files.
- Also, you could but the struct together with the function prototypes associated with manipulating it into the same header file.
- But there is no direct mechanism to restrict the manipulation of the structs internals to only be via those functions.
- To replace implementations we have to "throw away" our old .c file and replace it with a new one.
- There is no direct mechanism to incrementally extend a given struct and replace some of the implementation, thereby promoting code reuse.
- So it seems reasonable to try to create a preprocessor for C to try to add these and other features.

Object-Orientation Motivated by ADTs

- Roughly, what C++ tries to do is add these kind of features to C. This approach originated with Simula.
- We could map our understanding of how these new features work into C, by imagining we are writing a preprocessor for C++.
- We are supposed to do this in HW3.
- We can imagine methods being added to structs as functions pointers.
- Implementations of methods are just functions definitions where we add a parameter for our struct called this.
- When we declare a new instance of a class, we make sure to bind the associated functions pointers.
- To extend a class we might nest structs and add associated bindings.
- Virtual functions could be done by modifying our function pointer bindings.
- Our preprocessor could do checks to verify the private keyword was used correctly, etc.

Message Passing and OO

- Often complex systems are modeled as a collection of actors who communicate with each other.
- This often arises for instance in networking where one has two machines communicating, in parallel processing where one might have communicating processors, in OO where we might have object that invoke each others methods.
- You can think of the communication as calling some function and having a slot in that function that says the message type and then using maybe stdarg.h to pass some number of additional arguments along with this message:

```
void my_class(int msg_code, …);
```

- But how do we remember the state of our actor??

Preserving State the Fortran Way

- Since Fortran, you can preserve state in a functions.
- In C, this is done using the keyword static.
- So we have the following skeleton:

```
void my_class(int msg_code, …)
{ static int my_field1; //… other fields
  switch(msg_code) {
       case method1:
       break;
      …
       default:
  }
}
```

But Every Class Would be a Singleton

- The problem with the above is one function can only store one objects worth of data.
- How can we get around this?

```
void my_class(int *object_id, int msg_code, …)
{ static int *my_field_1;
  switch(msg_code) {
   case NEW:
      //my_field_i is a dynamically allocated array
     // managed by NEW and DESTROY
     // object_id's value is one index in this array.
 }
}
```

Some Differences With Between Message Passing and the ADT Style

- Notice in the ADT style, for each new class we have essentially created a new type. We could do a typedef on our struct as in HW description.
- In some languages such as Smalltalk, everything is an object, and has its own type.
- In the message passing way, we have new functions for each class, but its static members aren't really organized into a new type.

Objective-C

- Is there a language that actually takes the message passing approach?
- Yes. Objective-C
- Method calls look like:

```
[obj method:parameter];
```

- We'll talk about it more after the midterm.