Object Oriented Programming from Procedural Programming with a little Computational Thinking

Greg Michaelson

1. Introduction

In these notes, I'm going to look at how we might elaborate object oriented (OO) programming concepts by:

- starting with a procedural program to manipulate a bounded push down stack;
- successively applying the Computational Thinking (CT) stages of *pattern; identification* and *abstraction* to both the data structure and the sub-programs.

I assume that you're familiar with a push down stack and comfortable with arrays, records and sub-programs.

I'm going to use the Haggis reference language which I trust will be self-explanatory.

I'd like to thank the participants at the Haggis Workshop on 11th May 2015 at the University of Strathclyde, with whom this material was first developed.

If you spot any mistakes, please let me know!

Greg Michaelson
School of Mathematical and Computer Sciences, Heriot-Watt University

G.Michaelson@hw.ac.uk

2. The stack

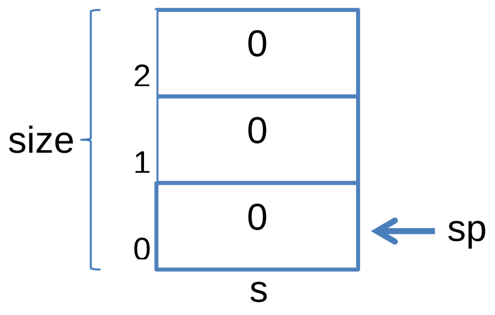
Suppose we want to make an integer stack from a:

- size;
- array of that size;
- stack pointer

For a new stack, we want every element set to 0, and the stack pointer to be set to 0 to indicate the bottom of the stack:

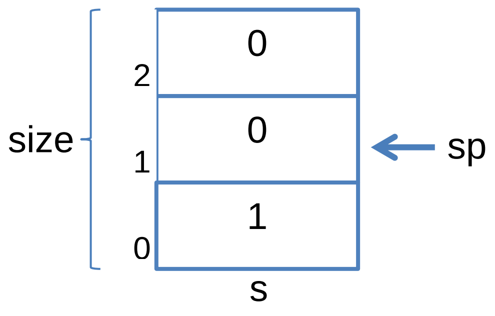
```
DECLARE size INITIALLY 3
DECLARE s IS ARRAY OF INTEGER INITIALLY [0]*size
DECLARE sp INITIALLY 0
```

We can visualise the initial stack as:

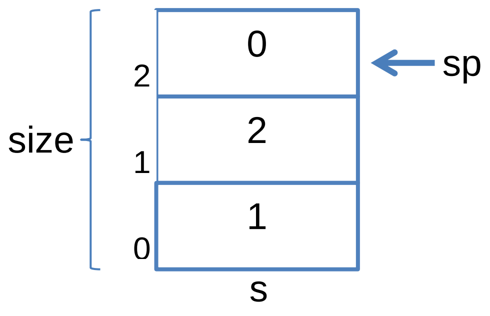


We can define the push operation as:

```
PROCEDURE push(INTEGER v)
  IF sp=size THEN
    <stack overflow action>
  ELSE
    SET s[sp] TO v
    SET sp TO sp+1
  END IF
END PROCEDURE
```



push(1)

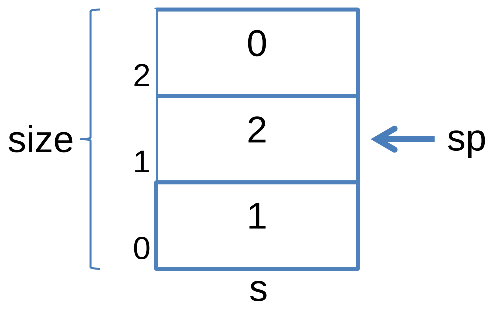


push(2)

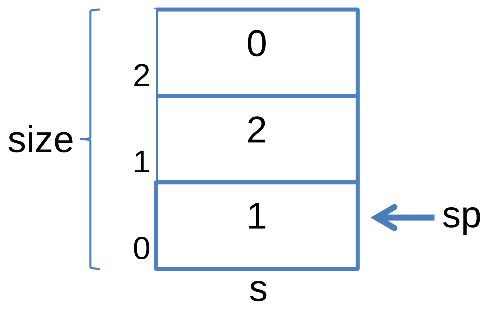
And we can define the pop operation as:

```
FUNCTION pop() RETURNS INTEGER
  IF sp=0 THEN
    <stack underflow action>
  ELSE
    SET sp TO sp-1
    RETURN s[sp]
  END IF
END FUNCTION
```

so:



SEND pop() TO DISPLAY ➔ 2



SEND pop() TO DISPLAY ➔ 1

This code has the apparent advantage that the variables are hard coded into the sub-programs that manipulate them, emphasising the strong conceptual connection between them.

However, the big disadvantages are that:

- the sub-programs can't be used with different variables representing other stacks;
- the variables are global to the whole program, so arbitrary code can change them with unpredictable effects.

2. Two stacks

Suppose we now want another stack but this time of size 4. We could just cut and paste the above code, rename all the variables and sub-programs, and change the initialisation:

```
DECLARE size1 INITIALLY 4
DECLARE s1 IS ARRAY OF INTEGER INITIALLY [0]*size1
DECLARE sp1 INITIALLY 0

PROCEDURE push1(INTEGER v)
  IF sp1=size1 THEN
    <stack overflow action>
  ELSE
    SET s1[sp1] TO v
    SET sp1 TO sp1+1
  END IF
END PROCEDURE

FUNCTION pop1() RETURNS INTEGER
  IF sp1=0 THEN
    <stack underflow action>
  ELSE
    SET sp1 TO sp1-1
    RETURN s1[sp1]
  END IF
END FUNCTION
```

Well, we certainly have two stacks. Alas:

- our code is every bit as insecure as before;
- and we now have twice as much of it;
- we have to explicitly call different sub-programs to manipulate different stacks.

3. Patterns and abstraction

Let's compare our two chunks of code and look for patterns by identifying differences. First of all, for the declarations, we have a common pattern:

```
DECLARE ? INITIALLY ?
DECLARE ? AS ARRAY OF INTEGER INITIALLY [0]*?
DECLARE ? INITIALLY 0
```

Conceptually, these three variables are strongly related in our stack model so we could abstract by:

- separating out the variables from their initialisations;
- defining a unitary record structure:

```
RECORD stack IS {INTEGER size,
                 ARRAY OF INTEGER s,
                 INTEGER sp}
```

- initialising record values when they're created:

```
DECLARE s1 IS stack(3,[0]*3,0)
DECLARE s2 IS stack(4,[0]*4,0)
```

Now we distinguish the variables:

```
s1.size, s1.s & s1.sp
```

from:

```
s2.size, s2.s & s2.sp
```

Similarly, for the sub-programs, we have common patterns:

```
PROCEDURE ?(INTEGER v)
  IF ?=? THEN
    <stack overflow action>
  ELSE
    SET ?[?] TO v
    SET ? TO ?+1
  END IF
END PROCEDURE

FUNCTION ?() RETURNS INTEGER
  IF ?=0 THEN
    <stack underflow action>
  ELSE
    SET ? TO ?-1
    RETURN ?[?]
  END IF
END FUNCTION
```

So, just as we abstracted our declarations with a record, we can do the same here by:

- introducing record formal parameters:
- abstracting inside the sub-programs with variable references relative to that record:

PROCEDURE push(*stack st*, INTEGER v)

```
    IF st.sp=st.size THEN
      <stack overflow action>
    ELSE
      SET st.s[st.sp] TO v
      SET st.sp TO st.sp+1
    END IF
  END PROCEDURE

  FUNCTION pop(stack st) RETURNS INTEGER
    IF st.sp=0 THEN
      <stack underflow action>
    ELSE
      SET st.sp TO st.sp-1
      RETURN st.s[st.sp]
    END IF
  END FUNCTION
```

Now, we can push to and pop from our two stacks with:

```
push(s1,value) ...pop(s1)...
```

and

```
push(s2,value) ...pop(s2)...
```

We seem to have solved our code bloat problem but we still have insecure code as the record variables are global. Furthermore, we've lost that strong connection between the stack information and the sub-programs that manipulate it.

4. Class = encapsulate(record + sub-programs)

It would be nice if we could somehow bundle together the record that holds the stack variables with the sub-programs that manipulate them, so that when we create a stack:

- the sub-programs know that they are only to work on the corresponding variables;
- it isn't possible to change those variables other than by using the sub-programs.

Before we see how to do this, let's think again about the record definition:

```
RECORD stack IS {INTEGER size,
                 ARRAY OF INTEGER s,
                 INTEGER sp}
```

Remember that this doesn't actually declare anything. Rather, it's a *specification* of what a record value contains. We can think of this as being like an architectural design for a house, which we certainly can't live in, unless it's 1:1 scale and made of tent cloth, but we can use to make actual houses.

Then, when we declare a RECORD, for example:

```
DECLARE s1 INITIALLY stack(3,[0]*3,0)
```

we’re asking for a new individual stack record value to be created with the fields initialised to the given values. We call this individual value an *instance* of the record.

Note that we use the RECORD identifier as if it were the name of a function that when calls returns an appropriate value, so we term the identifier the *constructor*.

Just as a record is an abstraction for a related group of variables, a *clas*s is an abstraction for a related group of variables and sub-programs. Here the subprograms are called *methods*.

A class definition looks like a record definition with an additional section where the methods are defined, for example:

```
CLASS stack IS {INTEGER size,
                ARRAY OF INTEGER s,
                INTEGER sp}
METHODS

PROCEDURE push ... END PROCEDURE

FUNCTION pop ... END FUNCTION

END CLASS
```

Note that different people may refer to the variables of the class as *fields* or *attributes* or *class variables*.

Just as with records, a class definition doesn’t actually create anything. Rather, it specifies how to make individual instances of the class, termed *objects*.

For example:

```
DECLARE s3 INITIALLY stack(10,[0]*10,0)
```

makes a new stack object associated with the variable s3, with size set to 10, s to [0...0] and sp to 0.

However, unlike a record value, the fields of an object can only be accessed by the methods of the object. For example, in our program we cannot refer to s3.size or s3.s or s3.sp.That is, the fields are *private*.

Nonetheless, we can access the methods of the object to manipulate the fields, by referring to them via the associated variable name, just as we refer to the fields of records. That is, the methods are *public*.

For example:

```
s3.push(...)
```

indicates that the push method for the object associated with s3 is to be called. This is also known as *message passing* as it's as if we're asking the object to perform the required method.

We say that an object *encapsulates* variables and sub-programs. We have achieved both code reuse through object creation, and code security through private class variables.

5. Madness in the methods

Let's now return to the methods. First of all, we no longer need to pass the class variables as parameters. Any variable mentions inside a method can only be to class variables, if not to formal parameters of local variables.

However, we don't just use the class variables themselves. Rather, we make use of the generic class variable THIS which always refers to the current object.

Thus, in our stack example, we might write

```
CLASS stack IS {INTEGER size,
                ARRAY OF INTEGER s,
                INTEGER sp)
METHODS

PROCEDURE push(INTEGER v)
  IF THIS.sp=THIS.size THEN
    <stack overflow action>
  ELSE
    SET THIS.s[THIS.sp] TO v
    SET THIS.sp TO THIS.sp+1
  END IF
END PROCEDURE

FUNCTION pop() RETURNS INTEGER
  IF THIS.sp=0 THEN
    <stack underflow action>
  ELSE
    SET THIS.sp TO THIS.sp-1
    RETURN THIS.s[THIS.sp]
  END IF
END FUNCTION
END CLASS
```

So when we call, say:

```
s3.push(42)
```

it’s as if we’ve replaced every occurrence of THIS in push with s3.

Frankly, I was alarmed when I first saw this style of coding as I’d been long used to structured programming where it’s deemed wrong for sub-programs to manipulate global variables. But, with encapsulation, this is wholly appropriate as only the methods of an object can change its class variables.

Anyway, let’s create another stack object:

```
DECLARE s4 INITIALLY stack(5,[0,0,0,0,0],0)
```

Now let’s push three values onto s3, and then pop them off and push them onto s4 in reverse order:

```
FOR i FROM 1 TO 3 DO
  s3.push(i)
END FOR
FOR i FROM 1 TO 3 DO
  s4.push(s3.pop())
END FOR
```

Notice that when we call s3.push(i) and s3.pop() we’re asking s3 to change its own s and sp, with reference to its own size. That is, for these calls THIS means s3. And when we call s4.push(i), we’re asking s4 to change its own s and sp, again with reference to its own size. That is, for this call THIS means s4.

6. Overloading constructors

Right now, when we make a stack, we have to explicitly nominate the size, the initial stack contents, and the initial stack pointer. We do this by calling the implicit constructor via the class identified. But we said we’d like every stack element to be initialised to 0 and to start with the stack pointer set to 0, so it would be nice if we could just supply the stack size and have standard code to set the stack contents and pointer.

We can define an explicit constructor:

```
CONSTRUCTOR (INTEGER sz)
  DECLARE THIS.size INITIALLY sz
  DECLARE THIS.s INITIALLY [0]*size
  DECLARE THIS.sp INITIALLY 0
END CONSTRUCTOR
```

In general, we can have multiple constructors without ambiguity provided they can be distinguished by the number and/or types of formal parameters. Here we are said to have *overloaded* the constructor. Note that the implicit constructor is still valid.

Note that an overloaded constructor:

- must declare and initialise all the class variables;
- doesn't have an identifier.

Now, we can create a stack with:

```
DECLARE s5 INITIALLY stack(30)
```

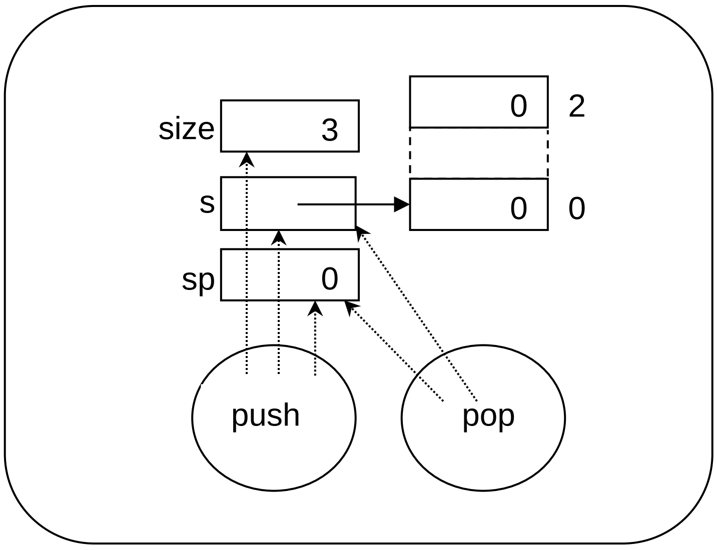
which will have the same effect as:

```
DECLARE s5 INITIALLY stack(30,[0]*30,0)
```

7. Summary

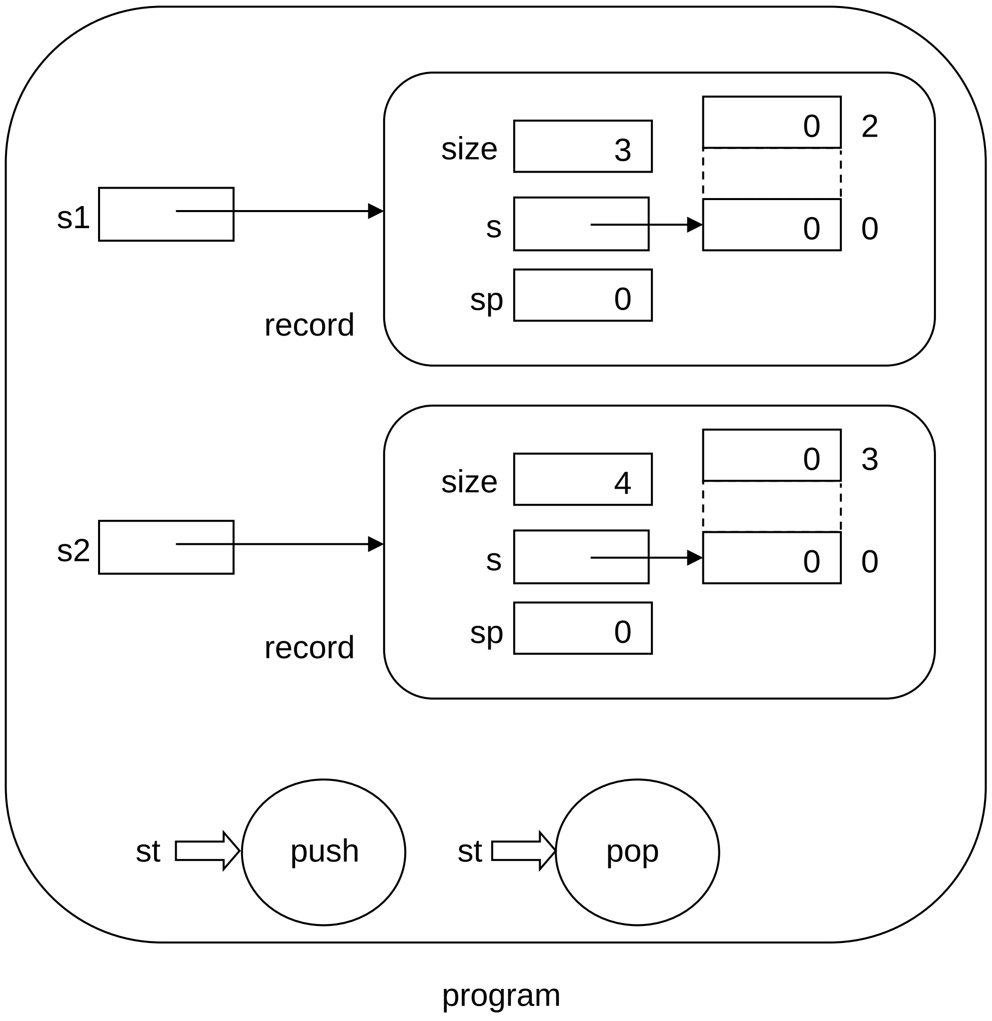
Let's draw things together by thinking about how we've represented stacks at each stage.

We started with global variables manipulated directly by global sub-programs:

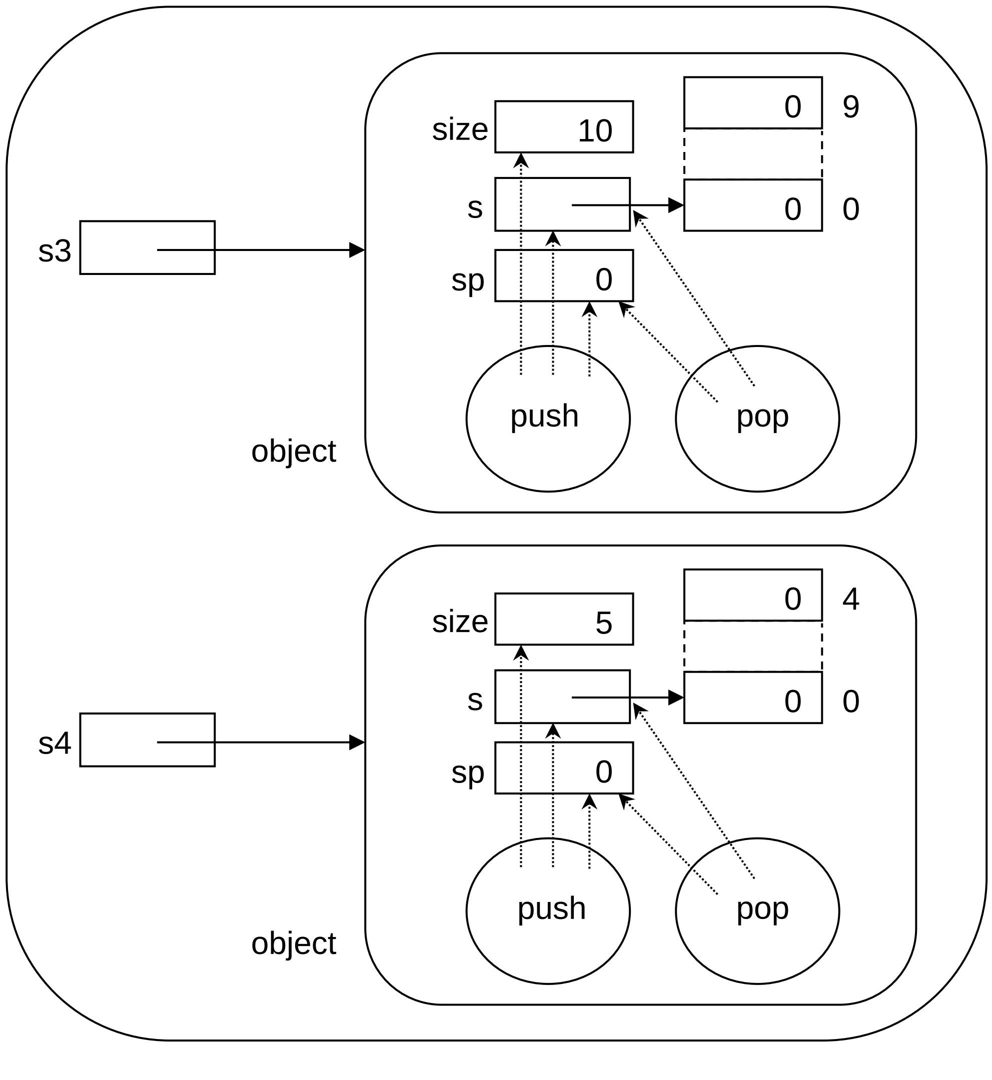


program

Next, we introduced records accessed by global variables, manipulated indirectly as parameters by global sub-programs:



Finally, we encapsulated the record and sub-programs to give objects accessed by global variables, with local variables manipulated directly by local methods.



Note how the objects in the final stage have the same structure as the whole program in the initial stage.