Chapter 13

Object−oriented programming

Object orientation is in many ways a natural style of programming whose techniques are reinvented constantly by every programmer [Coplien, 1992 #670]. Object notation consolidates these techniques so that much of the tedious programming necessary to use them is automatically handled by the interpreter. Objects in `hoc` can be thought of as a kind of abstract data type that is very useful in separating the idea of *what a thing does* from the details of *the way it goes about doing it*. Support for objects in `hoc` came late to NEURON, after the notion of cable sections, and as a consequence there are several types of variables (e.g. sections, mechanisms, range variables) that are clearly treated as objects from a conceptual point of view but grew up without a uniform syntax.

In `hoc`, an object is a collection of functions, procedures, and data; the data defines the state of the object. There is just enough extra syntax in `hoc` to support a subset of the object oriented programming paradigm. A subset, because it supports information hiding and polymorphism, but not inheritance. Nevertheless, it offers greatly increased ability to maintain conceptual control of the entire program. It immediately gives one all the power of data structures of languages such as C or Pascal, and most of the power of modules.

Object vs. class

First let's clarify the distinction between *object* and *class*. You're close to the mark if you think of a class as a cookie cutter that cuts out objects called cookies. A class is a general type, whereas an object of the class is a specific instance of the type. The idea of a class as a template motivated the keyword that signals the definition of classes in `hoc`: one surrounds a collection of functions, procedures, and variables with the keywords `begintemplate` and `endtemplate`.

From the user's point of view it is necessary to discuss how to create and destroy objects; what is an object reference; how to call an object's methods or access its data; and how to pass objects to functions. From the programmer's point of view it is necessary to discuss how to define a class. We'll give a general overview first before plunging into details.

The object model in `hoc`

The object model used by `hoc` manipulates references to objects and never the objects themselves. That is, object references are equivalent to pointers. The reference can be considered to be a label or alias for the object. Thus assignment

```
ob1 = ob2
```

means that `ob1` refers to the same object referred to by `ob2` and NOT that a new object is cloned from `ob2` and pointed to by `ob1`. Thus if `ob2`'s object contains a variable called `data` and that value is changed by the statement

```
ob2.data = 5
```

then

```
ob1.data
```

will print the value

```
5
```

It quickly becomes tedious to always talk about "the object referred to by `xxx`" so we often shorten the phrase to "`xxx`", always recalling that `xxx` is only one of possibly many labels for the object that it points to. In the next few paragraphs we'll strictly maintain the distinction between object reference and object, but be aware that we don't always exert such discipline.

Objects and object references

Declaring an object reference

Just as it is often convenient to deal with variables that can take on different number values (algebra is more powerful than arithmetic), it is often convenient to deal with object references that can refer to different objects at different times. Object references are declared with

```
objref name1, name2, name3, . . .
```

After an object reference has been declared, it refers to the `NULLobject` until it is associated with some other object (see below). The deprecated keyword `objectvar` is a synonym for `objref` that may be found in older programs; `objref` emphasizes the pointer nature of object references and has the advantage of being easier to type.

Creating and destroying an object

One creates an object with the `new` keyword. Thus

```
objref g
g = new Graph()
```

uses the `Graph` template to create one `Graph` object that we can refer to as `g`. We'll talk about where the templates come from later. Executing these two statements will create one graph window on the screen.

Several object references can refer to the same object. Continuing with the present example,

```
objref h
h = g
```

does not create a second graph but merely associates `h` with the same `Graph` object as `g`.

An object is destroyed when no object reference points to it. In this example, we can break the association between `g` and the `Graph` object by redeclaring `g`

```
objref g
```

so that `g` once again points to the `NULLobject`. However, the graph will persist on our screen because it is still referenced by `h`. To get rid of the graph we have to break this final reference, e.g. with the statement

```
h = g
```

Using an object reference

The object reference `g` should be thought of as pointing to an actual object located in the computer. This object has hidden variables that describe its state, along with visible variables, functions and procedures that do things to itself and to the outside world. The syntax for using the visible components of an object employs a "dot" notation reminiscent of how one accesses an element of a structure in C, e.g.

```
g.erase()
```

Of course, in C the object reference is really a pointer so one would use the arrow notation `a->b`. In C++, the object reference has the same syntax as a reference variable.

The `Graph` class is a built−in template with a large number of functions available for constructing x−y graphs.

```
g.size(0,1,0,10)
g.xaxis()
```

defines a model coordinate system and draws an axis in the third `Graph` window.

Defining an object template

The syntax for object templates is

```
begintemplate classname
public name1, name2, name3, . . .
external variable1, string2, function3, template4, . . .
   . . . hoc code . . .
endtemplate classname
```

where `classname` is the name of the class that the template defines. The `hoc` code can be almost anything you like, but generally it consists of declarations of variables and definitions of procedures and functions. Another term for a function or procedure that is defined in a class is a *method*.

Outside the template you cannot refer to any variable or functions except those listed in a `public` statement. Inside the template you cannot refer to any user−defined global variables or functions except those that appear in an `external` statement. However, you can execute built−in functions such as `printf()` and `exp()`, and you can also create objects from any externally−defined template.

Direct commands

Direct commands within a template, e.g.

```
begintemplate Foo
   public a
   a = 5      // this is a direct command
endtemplate Foo
```

are executed once when the template is interpreted. This means that declarations such as `double`, `strdef`, `func`, `xopen(file)`, etc., that need to be executed only once and not for each object are useful as direct commands. However, direct statements such as `a = 5` are less useful, since the value of `a` is lost when an actual object is created because the assignment statement is not executed at that time. Thus if we create a new object of class `Foo` named `footest`

```
oc>objref footest
oc>footest = new Foo()
oc>footest.a
        0
oc>
```

we see that the value of `footest.a` is 0, not 5.

Initializing variables in an object

To initialize variables to values other than 0, the template must contain an `init()` procedure. This procedure will be executed automatically every time a new object is created. If `init()` appears in the `public` list, you can execute it explicitly as well. For example, if we define a new class `Foo2` as

```
begintemplate Foo2
   public init, a
   proc init() {
      a = 5
   }
endtemplate Foo2
```

and then create a new object of this class

```
oc>objref foo2test
oc>foo2test = new Foo2()
```

now we find that `foo2test.a` has the nonzero value that we wanted

```
oc>foo2test.a
        5
oc>
```

Furthermore, if we assign a different value to `footest.a`

```
oc>foo2test.a = 6
oc>foo2test.a
        6
oc>
```


we can restore the original value by invoking `foo2test.init()`

```
oc>foo2test.init()
        0
oc>foo2test.a
        5
oc>
```

Keyword names

One restriction on templates is that `hoc` keywords cannot be redefined. This is an artifact of the order in which symbol tables are searched. For an example of how this affects programming, suppose we wanted to add a method to our `Stack` class that would print the name of every object in the stack. It might seem reasonable do this by inserting

```
proc print() {local cnt, i
   cnt = list.count()
   if (cnt == 0) {
      print "stack is empty"
   } else {
      for i=0,cnt-1 print list.object(i)
   }
}
```

into the body of the template and adding `print` to the `public` statement. This would allow us to call our new method with the highly mnemonic statement `stack.print()`. But when the interpreter tried to translate this to intermediate code, it would issue the error message

```
nrniv: parse error in stack3.hoc near line 2
   public push, pop, print
                          ^
```

and we would have to change the name of the method to something else, e.g. `printnames`.

Object references vs. object names

Up to this point we have been using *object references* to refer to objects, emphasizing the difference between an object itself and what we call it. Actually, each object does have a unique name that can be used anywhere a reference to the object is used. But these unique names are primarily intended for use by the library routines that construct NEURON's graphical interface, and while it may occasionally be useful for diagnostic or didactic purposes, it should never be used in ordinary programming. Object names are not guaranteed to be the same between different sessions of NEURON unless the sequence of creation and destruction of objects of the same type is identical. The object name is defined as `classname[index]` where the "index" is automatically incremented every time a new instance of that class is created. Index numbers are not reused after objects are deleted except when there are no existing objects of that type; then the index starts over again at 0.

The reason why unique object names are allowed at all is because some objects, such as the `PointProcessManager`, should be destroyed when their window is dismissed. This could not happen if the interpreter had an `objref` to that object, since objects are destroyed only when the reference count goes to 0. Thus the idiom is to cause the `VBox` window itself to increment the reference count for the object (and decrement it when the window is dismissed, using the `VBox`'s `ref()` or `dismiss_action()` method). Now the `hoc objref` that holds the reference can safely discard it, and the object will not be immediately destroyed. But the consequence is that there is now no way to get to the object (or the objects it created) from the interpreter except to use the object name, e.g. there is no other way to graph one of the point process variables in the `PointProcessManager`.

An example of the didactic use of object names

The name of an object can be used in any context in which a string is expected, e.g. a `print objref` statement. For example, if we execute the statements

```
objref g, h
g = new Graph()
h = g
```

then we see a graph on the computer screen, and

```
print g, h
```

returns

```
Graph[0] Graph[0]
```

because both `g` and `h` refer to the same `Graph` object. At this point if we type the command `print Graph[0]` we also get `Graph[0]`.

After redeclaring `g`

```
objref g
```

we find that `print g, h` gives us

```
NULLobject Graph[0]
```

Since one object reference (`h`) still points to `Graph[0]`, the graph is still visible, and `print Graph[0]` still produces `Graph[0]`.

Now asserting

```
h = g
```

discards the last reference to `Graph[0]`, destroying this object. Consequently the graph disappears from the screen, and `print g, h` produces

```
NULLobject NULLobject
```

Any lingering doubts concerning the fate of `Graph[0]` are dispelled when we find that `print Graph[0]` generates the error message

```
nrniv: Object ID doesn't exist: Graph[0]
 near line 11
print Graph[0]
               ^
```


Using objects to solve programming problems

Dealing with collections or sets

Most, if not all, nontrivial programming problems seem to involve the notion of a set or collection of objects. `hoc` can represent the concept of "more than one" in several ways, but the workhorses are the array of objects and the list of objects. The array is the most efficient but requires a prior knowledge of the number of objects to be stored. The list can store any number of objects at any time. This fact makes `List` the most often used class.

Arrays

Storage for an array of objects is declared with

```
objref array[size]
```

Only rarely is the size known when the program is written, so it is common practice to separate the declaration from the size definition and specify the latter just after the point in the execution when the size is finally known, as in

```
objref array[1] // must be declared even if size is wrong
proc set_size() {
   objref array[$1]
}
```

After the size is set, it can no longer be changed without redeclaring the entire array, which discards the references to any objects referenced in its previous incarnation. When an array is declared, all its elements reference the `NULLobject`.

The array is a random access object, which means that when the index is known, the object element can be evaluated or assigned. For example an array of five graphs can be created with

```
objref graphs[5]
for i=0, 4 { graphs[i] = new Graph() }
```

The internal name of each item in the array can be printed in reverse order with

```
for (i=4; i >= 0; i -= 1) { print graphs[i] }
```

Suppose we wanted to destroy the third (index = 2) graph. We can't simply say

```
objref graphs[2]
```

because this would discard the entire array, throwing away all of our graphs and creating a new array whose elements all point to the `NULLobject`. Instead, the way to make the reference count for the third graph become 0 is

```
objref nil
graphs[2] = nil
```

Example: emulating an "array of strings"

Even very simple templates have their uses. There is no such thing in `hoc` as an array of strings, but consider


```
begintemplate String
   public s
   strdef s
endtemplate String
```

Now you can use arrays of objects to get the functionality of arrays of strings.

```
objref s[3]
for i=0,2 s[i] = new String()  // they all start out empty
s[0].s = "hello"
s[2].s = "goodbye"
```

It is important to realize that there is no conflict between the use of `s` as the name of a `strdef` inside the template and the use of `s` as the name of an object reference outside the template.

Lists

A list is declared with the idiom

```
objref list
list = new List()
```

and objects are added to the list with the `append()` method, as in

```
for i=0, 4 { list.append(new Graph()) }
```

Notice that we do not have to know how many items will be added to the list before we start adding them. The `count()` method always returns the number of objects in the list and the `object()` method returns the item. One can print the names of the objects in a list with the statement,

```
for i=0, list.count - 1 { print list.object(i) }
```

Probably the most commonly used idiom in programming is iteration over a list, where each item in turn is processed by temporarily assigning it to an `objref`, as in

```
objref tobj
for i=0, list.count - 1 {
   tobj = list.object(i)
   // process the object referenced by tobj
}
objref tobj // only the list holds a reference to the last one
```

Example: a stack of objects

This template defines a class that can be used to create stacks of objects.

```
begintemplate Stack
   public push, pop
   objref list

   proc init() {
      list = new List()
   }

   proc push() {
      list.append($o1)
   }
```



```
        proc pop() {local cnt
            cnt = list.count()
            if (cnt == 0) {
                print "stack underflow"
                stop
            }
            $o1 = list.object(cnt-1)
            list.remove(cnt-1)
        }
    endtemplate Stack
```

After `hoc` parses this template, the statements

```
objref stack
stack = new Stack()
```

create an object that functions as a stack. At the time this new object is created, its `init()` procedure is executed, which creates an empty list for use by the `push()` and `pop()` procedures. Suppose we already have three `Graph` objects `g[0]`, `g[1]`, and `g[2]` (see **Creating an object** under **Objects and object references** above). Then `stack.push(g[1])` adds a reference to the second `Graph` at the end of the `Stack` object's internal list. `stack.pop(g[2])` would cause `g[2]` to reference the same object as `g[1]` and remove it from the stack.

In this example, we have exploited an existing object class (`List`) to create a new object class (`Stack`) that can be used to hold a stack of objects of any class we like−−not just objects of any of NEURON's built−in classes, but also objects of any other classes that we might dream up in the future! Note the use of the `List` class's `count()` and `remove()` methods to find the object at the end of the list and to remove this reference from the list.

Encapsulating code

Suppose you have a `hoc` file that works perfectly all by itself (when nothing else is loaded) and does something meaningful when you type `run()` at the `oc>` prompt. Also suppose the file has no direct commands except declarations (if it does have direct commands, just collect them into an `init()` procedure). Then, if you put the these lines at the beginning of the file

```
begin_template F1
public run
```

and this line at the end of the file

```
end_template F1
```

you have an object template. You can create an object and run it with

```
objref f1
f1 = new(F1)
f1.run()
```

and you will get identical behavior as before. What's been gained? Well, you can do this to a bunch of files and load them all together and never worry about variable or function name clashes between files because nothing (except the object templates and specific object names) is global.

All variables that are not explicitly initialized in an `init()` procedure start off with a value of 0. Therefore all variables used by the template probably should be declared with direct assignment statements in an `init()` procedure to make sure they will have good default values. It is possible to declare a variable with an assignment statement in procedure `P1` and then use it in a `public` procedure `P2`, but be careful of the case when the user executes `P2` before executing `P1`. If this happens, the variable will have a value of 0.

Polymorphism and inheritance

A language supports polymorphism when it automatically does the right thing whether a function is called on the base class or on an object of a subtype. Since an object reference can refer to any type of object, `hoc`'s object model is polymorphic. Thus, if `A` and `B` are different classes but happen to have a method with the same name, e.g. `foo()`, then if `oref` refers to an instance of either `A` or `B`, one can say `oref.foo()` and the method of the particular object type will be called. For a concrete example, suppose we have defined several different classes of objects that generate specialized graphs called `BodePlot`, `PowerSpect`, and `CrossCorr`, and that each of these classes has its own `plot()` and `erase()` method. We can easily automate plotting and erasing if we declare

```
proc plotall() { local i
  for i = 0, glist.count()-1 glist.object(i).plot()
}

proc eraseall() { local i
  for i = 0, glist.count()-1 glist.object(i).erase()
}
```

and, every time we spawn a new instance of one of these classes, we append it to a `List` object, e.g.

```
objref mygraflist
glist = new List()
   . . .
objref bp, ps, cc
bp = new BodePlot()
glist.append(bp)
ps = new PowerSpect()
glist.append(ps)
cc = new CrossCorr()
glist.append(cc)
```

Now we can take care of all of these graphs at once by invoking `plotall()` or `eraseall()`.

Inheritance allows one to define many kinds of subclasses starting from a more abstract base class. It is useful in capturing the "IS A" relationship, and is most effective when the subtype "IS A" kind of base type, i.e. whenever a program uses an object of the base type then it would also make sense if it used an object of the subtype. People often (ab)use inheritance when the IS A relationship does not hold in order to conveniently reuse a portion of the base class. When one class is "ALMOST LIKE" another, and that other is ready and waiting to be used, it is tempting to inherit the whole behavior and replace only the parts that are different. It's best to avoid this practice and instead factor out the behavior common to both classes, placing it in a base class that can be inherited by both classes.

In `hoc`, inheritance can only be emulated by having the "subclass" instance create its "superclass" instance during initialization and supply stub methods for calling the public methods of the superclass. For example, consider the trivial `Base` class

```
begintemplate Base
public a, b
objref this

proc a() {
   printf ("inside %s.a()\n", this)
}

proc b() {
   printf("inside %s.b()\n", this)
}
endtemplate Base
```

Then the following will look like a subclass of `Base`, where we provide our own implementation of `a` and "inherit" the method `b`:

```
begintemplate Sub
public a, b
objref this, base

proc init() {
   base = new Base()
}

proc a() {
   printf("inside %s.a\n", this)
}

proc b() {
   base.b()
}

endtemplate Sub
```

