CS117, Introduction to Programming III, C++
Spring 1998
microMOO

Midterm Project, Due Monday May 11

In this project, I want you to implement a simple, single user version of Pavel Curtis' LambdaMOO (Curtis, 1997) MUD. Since your MUD will implement a subset of the full functionality of Curtis' MUD, I'll call it **microMOO**.

Most of what you'll be implementing in this assignment is described in the *LambdaMOO Programmer's Manual*. You should look at it since **microMOO** is based on LambdaMOO. An html version is available at the LambdaMOO ftp site as well as a postscript version if you'd like to get a printout.

Exercise 1: Getting Acquainted

Visit the original LambdaMOO MUD site (`<telnet://lambda.moo.mud.org:8888>`), play around for a little bit, and write down your impressions.

The Database

I'll refer to the collection of all **microMOO** objects as the database.

Exercise 2: The Object Model

The first thing to do is to implement the object model described in the programmer's manual. The base code I've provided includes a LMObject class which you may adapt to the above definition. To make things a little easier, two classes which inherit from LMObject have also been provided: LMPlayer and LMLocation. These two classes respectively define player and location types. The programmer's manual specifies that each LambdaMOO object has a set of *attributes*, *properties*, and *verbs*. In my class definition, these are all grouped together, so an LMObject should have the following members:

- *id*, a string. This will be a unique identifier for the object. Object identifiers are numbers preceded by the a `#`. So for example, `#23` is a valid object identifier.
- *parent*, a string. This is identifier of the object's parent.
- *children*, a list of strings. This is a list of identifiers for object's children; those objects for which this is object is their parent.
- *name*, a string. This is the name of the object.
- *owner*, a string. This should be the *identifier* for the object that owns the current object.
- *location*, a string. The identifier of the location where the object resides.
- *contents*, a list of strings. The objects that are inside this object.
- *verbs*, a list of identifiers for verbs that the object can execute. For example, each object may be able to execute the verb **say**. But we might expect for an object to have its own specialized **say** verb. For example, object `#23` might have its own special implmentation of this verb, for example **#23:say**. These strings will be used to look up an appropriate procedure to execute when the verb is called. More on this later.

- *other_properties*, this member is actually a collection of non-standard properties. This allows a user to specialize an object. For now, we'll say that this is a *map* that contains strings indexed by strings. This will allow a user to add properties like *color* with values like *red* to an arbitrary object.

As you'll note from the programmer's manual, each object is uniquely identified by a `#` sign followed by a number. So for example, the id of the 23rd object created should be `#23`. You'll probably want to define a global structure to store the objects, indexed by their ids. This structure is in effect the database.

One way to handle this is to establish a global counter that is incremented each time an object is created. In this way, you can create an object id by generating a string by appending `#` to the current value of the global counter.

When you start your **microMOO**, you'll want to create a base object. If you think of the parent-child tree for the **microMOO**, this is the object that should be at the root of the tree. You should also create an **LMPlayer** object that represents *you*. So for example, if you type

```
pickup #10
```

at the command line, this should have the effect of adding object `#10` to the *contents* list of your **LMPlayer** object.

You should also create an **LMLocation** object that is the default place where objects get placed once created.

In the code base I've provided, a class called LMDBObject can be used to encapsulate the database. Use and adapt this as you see fit.

Exercise 3: microMOO Interface

As a MUD administrator, you'll also want some way of interfacing with your system. That is, at a minimum, you want to be able to:

- Create an arbitrary object, and have it added to the database.
- Inspect an arbitrary object. That is, look inside it and see how the member values have been set.
- Edit an object. That is, change the member values of an object.
- Delete an object.
- See a listing of the objects inside of the database.
- Startup the **microMOO** *read-eval-print* loop.

To help out with this, I have provided a simple command line interface that allows you to select one of these options from a printed menu. A window oriented browser is also in the works.

In either case, you will need to add functionality to the command interface so that it can be used to inspect and make simple additions and changes in the **microMOO** database.

You should write a set of functions to implement the following LambdaMOO commands:

- `LMObject` **create**(`string` *parent-id*). This returns an LMObject with parent set to the object *referenced* by the *parent-id*. So for example

  ```
  (create ``#1'')
  ```

should return a new object whose parent is the object with id `#1`. And of course the newly created object will be a child of `#1`. So this means that a call to **create** should create a new object, add it to the database, and generate a unique id for it. You should also verify that the supplied parent id is valid. We'll also add some simple inheritance here. Each child inherits the *other-properties* property of its owner.

- `void` **recycle**(`string` *owner-id*). This deletes the referenced object.
- `void` **move**(`string` *what-id*, `string` *where-id*). Change the location of the object referenced by *what-id* to *where-id*. Note that you have to update the "location" property of the "what" object as well as the "contents" property of the "where" object.
- `void` **look**(`string` *who-id*), print what objects are located in the same location as object referenced by *who-id*.
- `void` **take**(`string` *who-id*, `string` *what-id*, `string` *whom-id*), this removes object *what-id* from the *contents* list of *whom-id* and places it in the *contents* list of *who-id*.
- `void` **say**(`string` *who-id*, `string` *what*), this prints out a message "*who-id* says *what*".
- `void` **pickup**(`string` *who-id*, `string` *what-id*), this places *what-id* in the *contents* list of *who-id*. Note that *who-id* and *what-id* have to be at the same location.
- void **show_properties**(`string` *obj-id*). This should print all properties of object *obj-id*.
- void **add_property**(`string` *obj-id*, `string` *property_name*, `string` *property_value*). Adds a property *property_name* to object *obj-id* with value *property_value*.
- void **delete_property**(`string` *obj-id*, `string` *property_name*). Deletes a property *property_name* from object *obj-id*.
- void **set_property**(`string` *obj-id*, `string` *property_name*, `string` *value*). Sets a property to object *obj-id*.

You're welcome to also implement these in terms of member functions on LMObjects. You should also implement *accessor functions* for the other default properties.

Exercise 4: Saving State

Secondly, you'll need some way of storing the objects that exist. Also, you'll need some way of *initializing* the database and also of *checkpointing*, or writing back to disk, the database.

There are unimplemented **write_db**(`string` *filename*) and **read_db**(`string` *filename*) functions that are linked to menu items. Using the object model that you have developed in the first exercise, write member functions that will read in a set of objects that have been stored on disk. You may choose any consistent method for storing the objects that you would like.

One possible strategy is to begin the description of the object with some delimiter string, write out the default properties (*i.e.* name – id), output some special delimiter sequence for the user defined properties, then end both.

So for example, let's assume we want to write out object `#29`. Let's say that `#29` is a room named "east room" and has added properties properties *height* and *width*. Let's say that there are two objects that are inside of the room, `#55` and `#87`. Let's also assume that "east room" is inside "palace" (object id `#15` say) and that the owner of "east room" is root (*i.e.* object `#1`). So one possible save representation would be:

```
::
``east room''
#1
#15
#55,#87
#29
---
``height'', ``10''
``width'', ``10''
---
::
```

This will be followed by the printed representation of other objects.

Verbs

Verbs in the LambdaMOO nomenclature are programs. For example, **say** is a verb, as is **move**. You'll want to add some way of handling simple verbs.

In the following exercises, you'll develop a simple evaluator loop by filling in the guts of the *command interpreter* option of the simple command interface that I've provided.

Exercise 5: A Verb Dispatch Table

In later exercises, you will have to write an interpreter for **microMOO** statements passed in at the command line, *e.g.*

```
move #10 to #27
```

where we'll assume that `#10` is a valid object which is not a location and that `#27` is a valid location. In this case, what you'd ideally like is the ability to store a function (*e.g.* in this case the **move** command you wrote as part of **Exercise 3**) and retrieve it using the name of the function. In other words, to be able to have the interpretation result in:

```
lookup_verb(``move'')(``#1'', ``#27'');
```

The `lookup_verb` command could be implemented easily enough using a *map*, but you would like it to actually return something like a pointer to the **move** function. One way of making this nice is to use *function objects*.

```
class verb{
 public:
  void operator () () {return;}};
class move_verb : public verb {
 public:
  void operator() (string & id_obj, string & id_to)
                 {move(lookup_obj(id_obj), lookup_obj(id_to));}
  };
...
move_verb mv;
...
add_verb(``move'', mv);
```

That is, class **move_verb** defines an operator function which takes the three arguments that you'd expect.

The added benefit of doing it this way is that when you define *new verbs* via the *.program* command (this will be in part 2), you can create a new function object that is initialized with the parse structure and this can be used in interpretation.

If you look at the class definition for the LMDBObj, you'll see that both **add_verb**, and **lookup_verb** are listed as member functions. There is a member **verbs** for this class also. You should first complete the implementation of these and then populate the **verbs** table with function objects for the functions you defined in **Exercise 3**.

Exercise 6: Command Line Interpreter

Implement a simple command interpreter loop for your **microMOO**. This should process the string that's read in by the **doInterpret()** command (see the code for details). The idea here is to go from text that is input in the command window, to execution of an appropriate piece of program code. To do this you'll implement a simple command line interpreter. This should be able to handle simple commands like:

```
move #10 to #27
```

Typing in this line should move object `#10` from its current location to location `#27`.

For **microMOO** a valid command takes one of three forms (this is in the programmer's manual):

```
verb
verb direct-object
verb direct-object preposition indirect-object
```

Here is a list of verbs that you should be able to handle.

- **say** *string*, prints out the *string* and a line feed. Here, when you do the evaluation, you want to make sure that *direct object* is treated appropriately.
- **take** *obj1* **from** *obj2*, remove *obj1* from the *contents* list of *obj2* and place it in *your contents* list. Remember from **Exercise 2** you should create a **LMPlayer** object that is understood to be *you*.
- **pickup** *obj1*, this should result in *obj1* being placed in your *contents* list if it's in the same location that *you* are.
- **look**, this will result in a print out of the objects in your current *location*.
- **move** *obj1* **to** *obj2*, as specified in the above examples.

So one approach would be to use an instance of **LMcommand** inside of **doInterpret()** that would first parse the input string. You want to write member function
parse(`string &` *input_string*) which takes the command line string as its argument. In my code, **LMcommand** has four slots, one of which always has to be filled (guess). The **parse** member function should result in these members being filled in appropriately. You can then call another member function for **LMcommand** called **eval** which uses the values stored in the members to evaluate the command.

References

Curtis, Pavel. (1997). *LambdaMOO Programmer's Manual* . Available at
ftp://ftp.lambda.moo.mud.org/pub/MOO/ProgrammersManual.html/.