LPC Intermediates

Written by Descartes of Borg
borg@hebron.connected.com

November 1993

Chapter 1: Introduction

1.1 LPC Basics

Anyone reading this textbook should either have read the textbook LPC Basics or be familiar enough with mud realm coding such that not only are they capable of building rooms and other such objects involved in area coding, but they also have a good idea of what is going on when the code they write is executing. If you do not feel you are at this point, then go back and read LPC Basics before continuing. If you do so, you will find that what you read here will be much more meaningful to you.

1.2 Goals of This Textbook

The introductory textbook was meant to take people new to LPC from knowing nothing to being able to code a nice realm on any LPMud. There is naturally much more to LPC and to LPMud building, however, than building rooms, armours, monsters, and weapons. As you get into more complicated concepts like guilds, or desire to do more involved things with your realm, you will find the concepts detailed in LPC Basics to be lacking in support for these projects. Intermediate LPC is designed to take you beyond the simple realm building process into a full knowledge of LPC for functioning as a realm builder on an LPMud. The task of mudlib building itself is left to a later text. After reading this textbook and working through it by experimenting with actual code, the reader should be able to code game objects to fit any design or idea they have in mind, so long as I have been successful.

1.3 An Overview

What more is there? Well many of you are quite aware that LPC supports mappings and arrays and have been asking me why those were not detailed in LPC Basics. I felt that those concepts were beyond the scope of what I was trying to do with that textbook and were more fitting to this textbook. But new tools are all fine and dandy, what matters, however, is what you can do with those tools. The goal of LPC Basics was to get you to building quality LPMud realms. Mappings and arrays are not necessary to do that. The goal of this book is to allow you to code any idea you might want to code in your area. That ability requires the knowledge of mappings and arrays.

Any idea you want to code in an LPMud is possible. LPC is a language which is amazingly well suited to this task. All that prevents you from coding your ideas is your knowledge of LPC or an inadequate mudlib or your mudÕs theme or administrative policies. This textbook cannot make the mudlib you are working with any better, and it cannot change the mud theme or the mudÕs administrative policies. Never once think that LPC is incapable of doing what you want to do. If your idea is prevented by administrative policies or themes, then it is simply not an idea for your current mud. If the mudlib is inadequate, talk to the people in charge of your mudlib about what can be done at the mudlib level to facilitate it. You would be surprised by what is actually in the mudlib you did not know about. More important, after reading this textbook, you should be able to read all of the mudlib code in your mudÕs mudlib and understand what is going on at each line in the mudlib code. You may not as yet be able to reproduce that code on your own, but at least you can understand what is going on at the mudlib level.

This textbook starts out with a discussion about what the LPMud driver is doing. One nice thing about this textbook, in general it is completely driver and mudlib independent (excepting for the Dworkin Game Driver). The chapter on the game driver does not get into actual implementation, but instead deals with what all game drivers basically do in order to run the mud.

Next I discuss those magic topics everyone wants to know more about, arrays and mappings. Mappings may be simultaneously the easiest and most difficult data type to understand. Since they are sort of complex arrays in a loose sense, you really need to understand arrays before discussing them. All the same, once you understand them, they are much easier than arrays to use in real situations. At any rate, spend most of your time working with that chapter, because it is probably the most difficult, yet most useful chapter in the book.

After that follows a brief chapter on the LPC pre-compiler, a tool you can use for sorting out how your code will look before it gets sent to the compiler. Despite my horrid intro to it here, this chapter is perhaps the easiest chapter in the textbook. I put it after the mappings and arrays chapter for exactly that reason.

Strings are re-introduced next, going into more detail with how you can do such things as advanced command handling by breaking up strings. Once you understand arrays fairly well, this chapter should be really simple.

The next chapter is the second most important in the book. It may be the most important if you ever intend to go beyond the intermediate stage and dive into mudlib coding. That chapter involves the complex ideas behind LPC inheritance. Since the goal of this textbook is not to teach mudlib programming, the chapter is not a detailed discussion on object oriented programming. Understanding this chapter, however, will give you some good insights into what is involved with object oriented programming, as well as allow you to build more complex objects by overriding functions and defining your own base classes.

Finally, the textbook ends with a simple discussion of code debugging. This is not an essential chapter, but instead it is meant as more of an auxiliary supplement to what the knowledge you have accumulated so far.

1.4 Not Appearing in This Textbook

Perhaps what might appear to some as the most glaring omission of this textbook is largely a political omission, shadows. Never have I ever encountered an example of where a shadow was either the best or most effecient manner of doing anything. It does not follow from that, however, that there are no uses for shadows. My reasoning for omitting shadows from this textbook is that the learner is best served by learning the concepts in this textbook first and having spent time with them before dealing with the subject of shadows. In that way, I feel the person learning LPC will be better capable of judging the merits of using a shadow down the road. I will discuss shadows in a future textbook.

If you are someone who uses shadows some or a lot, please do not take the above paragraph as a personal attack. There may be some perfectly valid uses for shadows somewhere which I have yet to encounter. Nevertheless, they are not the ideal way to accomplish any given task, and therefore they are not considered for the purposes of this textbook an intermediate coding tool.

I have also omitted discussions of security and object oriented programming. Both are quite obviously mudlib issues. Many people, however, might take exception with my leaving out a discussion of object oriented programming. I chose to leave that for a later text, since most area builders code for the creativity, not for the computer science theory. In both the intermediate and beginner textbooks, I have chosen only to discuss theory where it is directly applicable to practical LPC programming. For people who are starting out green in LPC and want to code the next great mudlib, perhaps theory would be more useful. But for the purposes of this book, a discussion of object oriented programming is simply a snoozer. I do plan to get heavy into theory with the next textbook.

1.5 Summary

LPC is not difficult to learn. It is a language which, although pathetic compared to any other language for performing most computer language tasks, is incredibly powerful and unequalled for the tasks of building an area in MUD type games. For the beginner, it allows you to easily jump in and code useful objects without even knowing what you are doing. For the intermediate person, it allows you to turn any idea you have into textual virtual reality. And for the advanced person, itÕs object oriented features can allow you to build one of the most popular games on the internet. What you can do is simply limited by how much you know. And learning more does not require a computer science degree.

Chapter 2: The LPMud Driver

2.1 Review of Basic Driver/Mudlib Interaction

In the LPC Basics textbook, you learned a lot about the way the mudlib works, specifically in relation to objects you code in order to build your realm. Not much was discussed about the interaction between the mudlib and the driver. You should know, however, that the driver does the following:

1) When an object is first loaded into memory, the driver will call create() in native muds and reset() in compat muds. A creator uses create() or reset() to give initial values to the object.
2) At an interval setup by the game administrator, the driver calls the function reset(). This allows the object to regenerate monsters and such. Notice that in a compat mud, the same function is used to set up initial values as is used to reset the room.
3) Any time a living object comes in contact with an object of any sort,

the driver calls init() in the newly encountered object. This allows newly encountered objects to give living objects commands to execute through the add_action() efun, as well as perform other actions which should happen whenever a living thing encounters a given object.
4) The driver defines a set of functions known as efuns which are available to all objects in the game. Examples of commonly used efuns are: this_player(), this_object(), write(), say(), etc.

2.2 The Driver Cycle
The driver is a C program which runs the game. Its basic functions are to accept connections from the outside world so people can login, interpret the LPC code which defines LPC objects and how they function in the game, and accept user input and call the appropriate LPC functions which match the event. In its most simplest essence, it is an unending loop.

Once the game has booted up and is properly functioning (the boot up process will be discussed in a future, advanced LPC textbook), the driver enters a loop which does not terminate until the shutdown() efun is legally called or a bug causes the driver program to crash. First off, the driver handles any new incoming connections and passes control of the connection to a login object. After that, the driver puts together a table of commands which have been entered by users since the last cycle of the driver. After the command table is assembled, all messages scheduled to be sent to the connection from the last driver cycle are sent out to the user. At this point, the driver goes through the table of commands to be executed and executes each set of commands each object has stored there. The driver ends its cycle by calling the function heart_beat() in every object with a heart_beat() set and finally performing all pending call outs. This chapter will not deal with the handling of connections, but instead will focus on how the driver handles user commands and heartbeats and call outs.

2.3 User Commands
As noted in section 1.2, the driver stores a list of commands for each user to be executed each cycle. The commands list has the name of the living object performing the command, the object which gave the living object that command, and the function which is to be executed in order to perform the command. The driver refers to the object which typed in the command as the command giver. It is the command giver which gets returned as this_player() in most cases.

The driver starts at the top of the list of living objects with pending commands, and successively performs each command it typed by calling the function associated with the command and passing any arguments the command giver gave as arguments to the function. As the driver starts with the commands issued by a new living object, the command giver variable is changed to be equal to the new living object, so that during the sequence of functions initiated by that command, the efun this_player() returns the object which issued the command.

Let's look at the command buffer for an example player. Since the execution of his last command, Bozo has typed "north" and "tell descartes when is the next reboot". The command "north" is associated with the function "Do_Move()" in the room Bozo is in (the command "north" is automatically setup by the set_exits() efun in that room). The command "tell" is not specifically listed as a command for the player, however, in the player object there is a function called "cmd_hook()" which is associated with the command "", which matches any possible user input.

Once the driver gets down to Bozo, the command giver variable is set to the object which is Bozo. Then, seeing Bozo typed "north" and the function "north" is associated with, the driver calls Bozo's_Room->Do_Move(0). An argument of 0 is passed to the function since Bozo only typed the command "north" with no arguments. The room naturally calls some functions it needs, all the while such that the efun this_player() returns the object which is Bozo. Eventually, the room object will call move_player() in Bozo, which in turn calls the move_object() efun. This efun is responsible for changing an object's environment.

When the environment of an object changes, the commands available to it from objects in its previous environment as well as from its previous environment are removed from the object. Once that is done, the driver calls the efun init() in the new environment as well as in each object in the new environment. During each of these calls to init(), the object Bozo is still the command giver. Thus all add_action() efuns from this move will apply to Bozo. Once all those calls are done, control passes back from the move_object() efun to the move_player() lfun in Bozo. move_player() returns control back to Do_Move() in the old room, which returns 1 to signify to the driver that the command action was successful. If the Do_Move() function had returned 0 for some reason, the driver would have written "What?" (or whatever your driver's default bad command message is) to Bozo.

Once the first command returns 1, the driver proceeds on to Bozo's second command, following much the same structure. Note that with "tell descartes when is the next reboot", the driver passes "descartes when is the next reboot" to the function associated with tell. That function in turn has to decide what to do with that argument. After that command returns either 1 or 0, the driver then proceeds on to the next living object with commands pending, and so on until all living objects with pending commands have had their commands performed.

2.4 The Efuns set_heart_beat() and call_out()
Once all commands are performed for objects with commands pending, the driver then proceeds to call the heart_beat() function in all objects listed with the driver as having heartbeats. Whenever an object calls the efun set_heart_beat() with a non-zero argument (depending on your driver, what non-zero number may be important, but in most cases you call it with the int 1). The efun set_heart_beat() adds the object which calls set_heart_beat() to the list of objects with heartbeats. If you call it with an argument of 0, then it removes the object from the list of objects with heartbeats.

The most common use for heartbeats in the mudlib is to heal players and monsters and perform combat. Once the driver has finished dealing with the command list, it goes through the heartbeat list calling heart_beat() in each object in the list. So for a player, for example, the driver will call heart_beat() in the player which will:
1) age the player
2) heal the player according to a heal rate
3) check to see if there are any hunted, hunting, or attacking objects

around
4) perform an attack if step 3 returns true.
5) any other things which need to happen automatically roughly every second

Note that the more objects which have heartbeats, the more processing which has to happen every cycle the mud is up. Objects with heartbeats are thus known as the major hog of CPU time on muds.

The call_out() efun is used to perform timed function calls which do not need to happen as often as heartbeats, or which just happen once. Call outs let you specify the function in an object you want called. The general formula for call outs is:

```
call_out(func, time, args);
```

The third argument specifying arguments is optional. The first argument is a string representing the name of the function to be called. The second argument is how many seconds should pass before the function gets called.

Practically speaking, when an object calls call_out(), it is added to a list of objects with pending call outs with the amount of time of the call out and the name of the function to be called. Each cycle of the driver, the time is counted down until it becomes time for the function to be called. When the time comes, the driver removes the object from the list of objects with pending call outs and performs the call to the call out function, passing any special args originally specified by the call out function.

If you want a to remove a pending call before it occurs, you need to use the remove_call_out() efun, passing the name of the function being called out. The driver will remove the next pending call out to that function. This means you may have some ambiguity if more than one call out is pending for the same function.

In order to make a call out cyclical, you must reissue the call_out() efun in the function you called out, since the driver automatically removes the function from the call out table when a call out is performed. Example:

```
void foo() { call_out("hello", 10); }

void hello() { call_out("hello", 10); }
```

will set up hello() to be called every 10 seconds after foo() is first called. There are several things to be careful about here. First, you must watch to make sure you do not structure your call outs to be recursive in any unintended fashion. Second, compare what a set_heart_beat() does when compared directly to what call_out() does.

set_heart_beat():
a) Adds this_object() to a table listing objects with heartbeats.
b) The function heart_beat() in this_object() gets called every single driver cycle.

call_out():
a) Adds this_object(), the name of a function in this_object(), a time delay, and a set of arguments to a table listing functions with pending call outs.

b) The function named is called only once, and that call comes after the specified delay.

As you can see, there is a much greater memory overhead associated with call outs for part (a), yet that there is a much greater CPU overhead associated with heartbeats as shown in part (b), assuming that the delay for the call out is greater than a single driver cycle.

Clearly, you do not want to be issuing 1 second call outs, for then you get the worst of both worlds. Similarly, you do not want to be having heart beats in objects that can perform the same functions with call outs of a greater duration than 1 second. I personally have heard much talk about at what point you should use a call out over a heartbeat. What I have mostly heard is that for single calls or for cycles of a duration greater than 10 seconds, it is best to use a call out. For repetitive calls of durations less than 10 seconds, you are better off using heartbeats. I do not know if this is true, but I do not think following this can do any harm.

2.5 Summary
Basic to a more in depth understanding of LPC is and understanding of the way in which the driver interacts with the mudlib. You should now understand the order in which the driver performs functions, as well as a more detailed knowledge of the efuns this_player(), add_action(), and move_object() and the lfun init(). In addition to this building upon knowledge you got from the LPC Basics textbook, this chapter has introduced call outs and heartbeats and the manner in which the driver handles them. You should now have a basic understanding of call outs and heartbeats such that you can experiment with them in your realm code.

Chapter 3: Complex Data Types

3.1 Simple Data Types
In the textbook LPC Basics, you learned about the common, basic LPC data types: int, string, object, void. Most important you learned that many operations and functions behave differently based on the data type of the variables upon which they are operating. Some operators and functions will even give errors if you use them with the wrong data types. For example, "a"+"b" is handled much differently than 1+1. When you ass "a"+"b", you are adding "b" onto the end of "a" to get "ab". On the other hand, when you add 1+1, you do not get 11, you get 2 as you would expect.

I refer to these data types as simple data types, because they atomic in that they cannot be broken down into smaller component data types. The object data type is a sort of exception, but you really cannot refer individually to the components which make it up, so I refer to it as a simple data type.

This chapter introduces the concept of the complex data type, a data type which is made up of units of simple data types. LPC has two common complex data types, both kinds of arrays. First, there is the traditional array which stores values in consecutive elements accessed by a number representing which element they are stored in. Second is an associative array called a mapping. A mapping associates to values together to allow a more natural access to data.

3.2 The Values NULL and 0

Before getting fully into arrays, there first should be a full understanding of the concept of NULL versus the concept of 0. In LPC, a null value is represented by the integer 0. Although the integer 0 and NULL are often freely interchangeable, this interchangeability often leads to some great confusion when you get into the realm of complex data types. You may have even encountered such confusion while using strings.

0 represents a value which for integers means the value you add to another value yet still retain the value added. This for any addition operation on any data type, the ZERO value for that data type is the value that you can add to any other value and get the original value. Thus: A plus ZERO equals A where A is some value of a given data type and ZERO is the ZERO value for that data type. This is not any sort of official mathematical definition. There exists one, but I am not a mathematician, so I have no idea what the term is. Thus for integers, 0 is the ZERO value since 1 + 0 equals 1.

NULL, on the other hand, is the absence of any value or meaning. The LPC driver will interpret NULL as an integer 0 if it can make sense of it in that context. In any context besides integer addition, A plus NULL causes an error. NULL causes an error because adding valueless fields in other data types to those data types makes no sense.

Looking at this from another point of view, we can get the ZERO value for strings by knowing what added to "a" will give us "a" as a result. The answer is not 0, but instead "". With integers, interchanging NULL and 0 was acceptable since 0 represents no value with respect to the integer data type. This interchangeability is not true for other data types, since their ZERO values do not represent no value. Namely, "" represents a string of no length and is very different from 0.

When you first declare any variable of any type, it has no value. Any data type except integers therefore must be initialized somehow before you perform any operation on it. Generally, initialization is done in the create() function for global variables, or at the top of the local function for local variables by assigning them some value, often the ZERO value for that data type. For example, in the following code I want to build a string with random words:

```
string build_nonsense() {
    string str;
    int i;

    str = ""; /* Here str is initialized to the string
ZERO value */
    for(i=0; i<6; i++) {
        switch(random(3)+1) {
            case 1: str += "bing"; break;
            case 2: str += "borg"; break;
            case 3: str += "foo"; break;
        }
        if(i==5) str += ".\n";
        else str += " ";
    }
    return capitalize(str);
}
```

If we had not initialized the variable str, an error would have resulted from trying to add a string to a NULL value. Instead, this code first initializes str to the ZERO value for strings, "". After that, it enters a loop which makes 6 cycles, each time randomly adding one of three possible words to the string. For all words except the last, an additional blank character is added. For the last word, a period and a return character are added. The function then exits the loop, capitalizes the nonsense string, then exits.

3.3 Arrays in LPC
An array is a powerful complex data type of LPC which allows you to access multiple values through a single variable. For instance, Nightmare has an indefinite number of currencies in which players may do business. Only five of those currencies, however, can be considered hard currencies. A hard currency for the sake of this example is a currency which is readily exchangeable for any other hard currency, whereas a soft currency may only be bought, but not sold. In the bank, there is a list of hard currencies to allow bank keepers to know which currencies are in fact hard currencies. With simple data types, we would have to perform the following nasty operation for every exchange transaction:

```
int exchange(string str) {
    string from, to;
    int amt;

    if(!str) return 0;
    if(sscanf(str, "%d %s for %s", amt, from, to) != 3)
      return 0;
    if(from != "platinum" && from != "gold" && from !=
      "silver" &&
      from != "electrum" && from != "copper") {
        notify_fail("We do not buy soft currencies!\n");
        return 0;
    }
    ...
}
```

With five hard currencies, we have a rather simple example. After all it took only two lines of code to represent the if statement which filtered out bad currencies. But what if you had to check against all the names which cannot be used to make characters in the game? There might be 100 of those; would you want to write a 100 part if statement? What if you wanted to add a currency to the list of hard currencies? That means you would have to change every check in the game for hard currencies to add one more part to the if clauses. Arrays allow you simple access to groups of related data so that you do not have to deal with each individual value every time you want to perform a group operation.

As a constant, an array might look like this:

```
    ({ "platinum", "gold", "silver", "electrum", "copper" })
```

which is an array of type string. Individual data values in arrays are called elements, or sometimes members. In code, just as constant

strings are represented by surrounding them with "", constant arrays are
represented by being surrounded by ({ }), with individual elements of
the array being separated by a ,.

You may have arrays of any LPC data type, simple or complex. Arrays
made up of mixes of values are called arrays of mixed type. In most
LPC drivers, you declare an array using a throw-back to C language
syntax for arrays. This syntax is often confusing for LPC coders
because the syntax has a meaning in C that simply does not translate into
LPC. Nevertheless, if we wanted an array of type string, we would
declare it in the following manner:

```
string *arr;
```

In other words, the data type of the elements it will contain followed by
a space and an asterisk. Remember, however, that this newly declared
string array has a NULL value in it at the time of declaration.

3.4 Using Arrays
You now should understand how to declare and recognize an array in
code. In order to understand how they work in code, let's review the
bank code, this time using arrays:

```
string *hard_currencies;

int exchange(string str) {
    string from, to;
    int amt;

    if(!str) return 0;
    if(sscanf(str, "%d %s for %s", amt, from, to) != 3)
return 0;
    if(member_array(from, hard_currencies) == -1) {
        notify_fail("We do not buy soft currencies!\n");
        return 0;
    }
    ...
}
```

This code assumes hard_currencies is a global variable and is initialized
in create() as:

```
    hard_currencies = ({ "platinum", "gold", "electrum", "silver",
   "copper" });
```

Ideally, you would have hard currencies as a #define in a header file for
all objects to use, but #define is a topic for a later chapter.

Once you know what the member_array() efun does, this method
certainly is much easier to read as well as is much more efficient and
easier to code. In fact, you can probably guess what the
member_array() efun does: It tells you if a given value is a member of
the array in question. Specifically here, we want to know if the currency
the player is trying to sell is an element in the hard_curencies array.
What might be confusing to you is, not only does member_array() tell us
if the value is an element in the array, but it in fact tells us which element
of the array the value is.

How does it tell you which element? It is easier to understand arrays if

you think of the array variable as holding a number. In the value above, for the sake of argument, we will say that hard_currencies holds the value 179000. This value tells the driver where to look for the array hard_currencies represents. Thus, hard_currencies points to a place where the array values may be found. When someone is talking about the first element of the array, they want the element located at 179000. When the object needs the value of the second element of the array, it looks at 179000 + one value, then 179000 plus two values for the third, and so on. We can therefore access individual elements of an array by their index, which is the number of values beyond the starting point of the array we need to look to find the value. For the array hard_currencies array:
"platinum" has an index of 0.
"gold" has an index of 1.
"electrum" has an index of 2.
"silver" has an index of 3.
"copper" has an index of 4.

The efun member_array() thus returns the index of the element being tested if it is in the array, or -1 if it is not in the array. In order to reference an individual element in an array, you use its index number in the following manner:
array_name[index_no]
Example:
hard_currencies[3]
where hard_currencies[3] would refer to "silver".

So, you now should now several ways in which arrays appear either as a whole or as individual elements. As a whole, you refer to an array variable by its name and an array constant by enclosing the array in ({ }) and separating elements by ,. Individually, you refer to array variables by the array name followed by the element's index number enclosed in [], and to array constants in the same way you would refer to simple data types of the same type as the constant. Examples:

Whole arrays:
variable: arr
constant: ({ "platinum", "gold", "electrum", "silver", "copper" })

Individual members of arrays:
variable: arr[2]
constant: "electrum"

You can use these means of reference to do all the things you are used to doing with other data types. You can assign values, use the values in operations, pass the values as parameters to functions, and use the values as return types. It is important to remember that when you are treating an element alone as an individual, the individual element is not itself an array (unless you are dealing with an array of arrays). In the example above, the individual elements are strings. So that:

```
    str = arr[3] + " and " + arr[1];
```

will create str to equal "silver and gold". Although this seems simple enough, many people new to arrays start to run into trouble when trying to add elements to an array. When you are treating an array as a whole and you wish to add a new element to it, you must do it by adding another array.

Note the following example:

```
string str1, str2;
string *arr;

str1 = "hi";
str2 = "bye";
/* str1 + str2 equals "hibye" */
arr = ({ str1 }) + ({ str2 });
/* arr is equal to ({ str1, str2 }) */
```

Before going any further, I have to note that this example gives an extremely horrible way of building an array. You should set it: arr = ({ str1, str2 }). The point of the example, however, is that you must add like types together. If you try adding an element to an array as the data type it is, you will get an error. Instead you have to treat it as an array of a single element.

3.5 Mappings

One of the major advances made in LPMuds since they were created is the mapping data type. People alternately refer to them as associative arrays. Practically speaking, a mapping allows you freedom from the association of a numerical index to a value which arrays require. Instead, mappings allow you to associate values with indices which actually have meaning to you, much like a relational database.

In an array of 5 elements, you access those values solely by their integer indices which cover the range 0 to 4. Imagine going back to the example of money again. Players have money of different amounts and different types. In the player object, you need a way to store the types of money that exist as well as relate them to the amount of that currency type the player has. The best way to do this with arrays would have been to store an array of strings representing money types and an array of integers representing values in the player object. This would result in CPU-eating ugly code like this:

```
int query_money(string type) {
    int i;

    i = member_array(type, currencies);
    if(i>-1 && i < sizeof(amounts))  /* sizeof efun
returns # of elements */
        return amounts[i];
    else return 0;
}
```

And that is a simple query function. Look at an add function:

```
void add_money(string type, int amt) {
    string *tmp1;
    int * tmp2;
    int i, x, j, maxj;

    i = member_array(type, currencies);
    if(i >= sizeof(amounts)) /*  corrupt data, we are in
      a bad way */
        return;
    else if(i== -1) {
        currencies += ({ type });
```

```
            amounts += ({ amt });
            return;
        }
        else {
            amounts[i] += amt;
            if(amounts[i] < 1) {
                tmp1 = allocate(sizeof(currencies)-1);
                tmp2 = allocate(sizeof(amounts)-1);
                for(j=0, x =0, maxj=sizeof(tmp1); j < maxj;
                  j++) {
                    if(j==i) x = 1;
                    tmp1[j] = currencies[j+x];
                    tmp2[j] = amounts[j+x];
                }
                currencies = tmp1;
                amounts = tmp2;
            }
        }
    }
```

That is really some nasty code to perform the rather simple concept of adding some money. First, we figure out if the player has any of that kind of money, and if so, which element of the currencies array it is. After that, we have to check to see that the integrity of the currency data has been maintained. If the index of the type in the currencies array is greater than the highest index of the amounts array, then we have a problem since the indices are our only way of relating the two arrays. Once we know our data is in tact, if the currency type is not currently held by the player, we simply tack on the type as a new element to the currencies array and the amount as a new element to the amounts array. Finally, if it is a currency the player currently has, we just add the amount to the corresponding index in the amounts array. If the money gets below 1, meaning having no money of that type, we want to clear the currency out of memory.

Subtracting an element from an array is no simple matter. Take, for example, the result of the following:

```
string *arr;

arr = ({ "a", "b", "a" });
arr -= ({ arr[2] });
```

What do you think the final value of arr is? Well, it is:

```
    ({ "b", "a" })
```

Subtracting arr[2] from the original array does not remove the third element from the array. Instead, it subtracts the value of the third element of the array from the array. And array subtraction removes the first instance of the value from the array. Since we do not want to be forced on counting on the elements of the array as being unique, we are forced to go through some somersaults to remove the correct element from both arrays in order to maintain the correspondence of the indices in the two arrays.

Mappings provide a better way. They allow you to directly associate the money type with its value. Some people think of mappings as arrays where you are not restricted to integers as indices. Truth is, mappings are an entirely different concept in storing aggregate information. Arrays force you to choose an index which is meaningful to the machine for locating the appropriate data. The indices tell the machine how many elements beyond the first value the value you desire can be found. With mappings, you choose indices which are meaningful to you without worrying about how that machine locates and stores it.

You may recognize mappings in the following forms:

```
constant values:
whole: ([ index:value, index:value ]) Ex: ([ "gold":10, "silver":20 ])
element:  10

variable values:
whole:    map   (where map is the name of a mapping variable)
element: map["gold"]
```

So now my monetary functions would look like:

```
int query_money(string type) { return money[type]; }

void add_money(string type, int amt) {
    if(!money[type]) money[type] = amt;
    else money[type] += amt;
    if(money[type] < 1)
      map_delete(money, type);          /* this is for
          MudOS */
            ...OR...
            money = m_delete(money, type)  /* for some
          LPMud 3.* varieties */
            ... OR...
         m_delete(money, type);    /* for other LPMud 3.*
          varieties */
}
```

Please notice first that the efuns for clearing a mapping element from the mapping vary from driver to driver. Check with your driver's documentation for the exact name an syntax of the relevant efun.

As you can see immediately, you do not need to check the integrity of your data since the values which interest you are inextricably bound to one another in the mapping. Secondly, getting rid of useless values is a simple efun call rather than a tricky, CPU-eating loop. Finally, the query function is made up solely of a return instruction.

You must declare and initialize any mapping before using it.
Declarations look like:

```
mapping map;
```

Whereas common initializations look like:

```
map = ([]);
map = allocate_mapping(10)   ...OR...   map = m_allocate(10);
map = ([ "gold": 20, "silver": 15 ]);
```

As with other data types, there are rules defining how they work in common operations like addition and subtraction:

```
    ([ "gold":20, "silver":30 ]) + ([ "electrum":5 ])
```

gives:

```
    (["gold":20, "silver":30, "electrum":5])
```
Although my demonstration shows a continuity of order, there is in fact
no guarantee of the order in which elements of mappings will stored.
Equivalence tests among mappings are therefore not a good thing.

3.6 Summary
Mappings and arrays can be built as complex as you need them to be.
You can have an array of mappings of arrays. Such a thing would be
declared like this:

```
mapping *map_of_arrs;
```
which might look like:
```
({ ([ ind1: ({ valA1, valA2}), ind2: ({valB1, valB2}) ]), ([ indX:
({valX1,valX2}) ]) })
```

Mappings may use any data type as an index, including objects.
Mapping indices are often referred to as keys as well, a term from
databases. Always keep in mind that with any non-integer data type,
you must first initialize a variable before making use of it in common
operations such as addition and subtraction. In spite of the ease and
dynamics added to LPC coding by mappings and arrays, errors caused
by failing to initialize their values can be the most maddening experience
for people new to these data types. I would venture that a very high
percentage of all errors people experimenting with mappings and arrays
for the first time encounter are one of three error messages:
```
        Indexing on illegal type.
        Illegal index.
        Bad argument 1 to (+ += - -=) /* insert your favourite operator */
```
Error messages 1 and 3 are darn near almost always caused by a failure
to initialize the array or mapping in question. Error message 2 is caused
generally when you are trying to use an index in an initialized array
which does not exist. Also, for arrays, often people new to arrays will
get error message 3 because they try to add a single element to an array
by adding the initial array to the single element value instead of adding
an array of the single element to the initial array. Remember, add only
arrays to arrays.

At this point, you should feel comfortable enough with mappings and
arrays to play with them. Expect to encounter the above error messages
a lot when first playing with these. The key to success with mappings is
in debugging all of these errors and seeing exactly what causes wholes
in your programming which allow you to try to work with uninitialized
mappings and arrays. Finally, go back through the basic room code and
look at things like the set_exits() (or the equivalent on your mudlib)
function. Chances are it makes use of mappings. In some instances, it
will use arrays as well for compatibility with mudlib.n.

Chapter 4: The LPC Pre-Compiler

4.1 Review
The previous chapter was quite heavy, so now I will slow down a bit so
you can digest and play with mappings and arrays by taking on the
rather simple topic of the LPC pre-compiler. By this point, however,
you should well understand how the driver interacts with the mudlib and
be able to code objects which use call outs and heart beats. In addition,
you should be coding simple objects which use mappings and arrays,

noting how these data types perform in objects. It is also a good idea to start looking in detail at the actual mudlib code that makes up your mud. See if you understand everything which is going on in your mudlibs room and monster codes. For things you do not understand, ask the people on your mud designated to answer creator coding questions.

Pre-compiler is actually a bit of a misnomer since LPC code is never truly compiled. Although this is changing with prototypes of newer LPC drivers, LPC drivers interpret the LPC code written by creators rather than compile it into binary format. Nevertheless, the LPC pre-compiler functions still perform much like pre-compilers for compiled languages in that pre-compiler directives are interpreted before the driver even starts to look at object code.

4.2 Pre-compiler Directives

If you do not know what a pre-compiler is, you really do not need to worry. With respect to LPC, it is basically a process which happens before the driver begins to interpret LPC code which allows you to perform actions upon the entire code found in your file. Since the code is not yet interpreted, the pre-compiler process is involved before the file exists as an object and before any LPC functions or instructions are ever examined. The pre-compiler is thus working at the file level, meaning that it does not deal with any code in inherited files.

The pre-compiler searches a file sent to it for pre-compiler directives. These are little instructions in the file meant only for the pre-compiler and are not really part of the LPC language. A pre-compiler directive is any line in a file beginning with a pound (#) sign. Pre-compiler directives are generally used to construct what the final code of a file will look at. The most common pre-compiler directives are:

```
#define
#undefine
#include
#ifdef
#ifndef
#if
#elseif
#else
#endif
#pragma
```

Most realm coders on muds use exclusively the directives #define and #include. The other directives you may see often and should understand what they mean even if you never use them.

The first pair of directives are:
```
#define
#undefine
```

The #define directive sets up a set of characters which will be replaced any where they exist in the code at precompiler time with their definition. For example, take:

```
#define OB_USER "/std/user"
```

This directive has the pre-compiler search the entire file for instances of

OB_USER. Everywhere it sees OB_USER, it replaces with "/std/user". Note that it does not make OB_USER a variable in the code. The LPC interpreter never sees the OB_USER label. As stated above, the pre-compiler is a process which takes place before code interpretation. So what you wrote as:

```
#define OB_USER "/std/user"

void create() {
    if(!file_exists(OB_USER+".c")) write("Merde! No user file!");
    else write("Good! User file still exists!");
}
```

would arrive at the LPC interpreter as:

```
void create() {
    if(!file_exists("/std/user"+".c")) write("Merde! No user file!");
    else write("Good! User file still exists!");
}
```

Simply put, #define just literally replaces the defined label with whatever follows it. You may also use #define in a special instance where no value follows. This is called a binary definition. For example:

```
#define __NIGHTMARE
```

exists in the config file for the Nightmare Mudlib. This allows for pre-compiler tests which will be described later in the chapter.

The other pre-compiler directive you are likely to use often is #include. As the name implies, #include includes the contents of another file right into the file being pre-compiled at the point in the file where the directive is placed. Files made for inclusion into other files are often called header files. They sometimes contain things like #define directives used by multiple files and function declarations for the file. The traditional file extension to header files is .h.

Include directives follow one of 2 syntax's:

```
#include
#include "filename"
```

If you give the absolute name of the file, then which syntax you use is irrelevant. How you enclose the file name determines how the pre-compiler searches for the header files. The pre-compiler first searches in system include directories for files enclosed in <>. For files enclosed in "", the pre-compiler begins its search in the same directory as the file going through the pre-compiler. Either way, the pre-compiler will search the system include directories and the directory of the file for the header file before giving up. The syntax simply determines the order.

The simplest pre-compiler directive is the #pragma directive. It is doubtful you will ever use this one. Basically, you follow the directive with some keyword which is meaningful to your driver. The only keyword I have ever seen is strict_types, which simply lets the driver know you want this file interpreted with strict data typing. I doubt you will ever need to use this, and you may never even see it. I just included it in the list in the event you do see it so you do not think it is doing anything truly meaningful.

The final group of pre-compiler directives are the conditional pre-compiler directives. They allow you to pre-compile the file one way given the truth value of an expression, otherwise pre-compile the file another way. This is mostly useful for making code portable among mudlibs, since putting the m_delete() efun in code on a MudOS mud would normally cause an error, for example. So you might write the following:

```
#ifdef  MUDOS
    map_delete(map, key);
#else
    map = m_delete(map, key);
#endif
```

which after being passed through the pre-compiler will appear to the interpreter as:

```
    map_delete(map, key);
```

on a MudOS mud, and:

```
    map = m_delete(map, key);
```

on other muds. The interpreter never sees the function call that would cause it to spam out in error.

Notice that my example made use of a binary definition as described above. Binary definitions allow you to pass certain code to the interpreter based on what driver or mudlib you are using, among other conditions.

4.3 Summary

The pre-compiler is a useful LPC tool for maintaining modularity among your programs. When you have values that might be subject to change, but are used widely throughout your files, you might stick all of those values in a header file as #define statements so that any need to make a future change will cause you to need to change just the #define directive. A very good example of where this would be useful would be a header file called money.h which includes the directive:
#define HARD_CURRENCIES ({ "gold", "platinum", "silver", "electrum", "copper" })
so that if ever you wanted to add a new hard currency, you only need change this directive in order to update all files needing to know what the hard currencies are.

The LPC pre-compiler also allows you to write code which can be ported without change among different mudlibs and drivers. Finally, you should be aware that the pre-compiler only accepts lines ending in carriage returns. If you want a multiple line pre-compiler directive, you need to end each incomplete line with a backslash(\).

Chapter 5: Advanced String Handling

5.1 What a String Is

The LPC Basics textbook taught strings as simple data types. LPC generally deals with strings in such a matter. The underlying driver program, however, is written in C, which has no string data type. The driver in fact sees strings as a complex data type made up of an array of characters, a simple C data type. LPC, on the other hand does not recognize a character data type (there may actually be a driver or two out there which do recognize the character as a data type, but in general not). The net effect is that there are some array-like things you can do with strings that you cannot do with other LPC data types.

The first efun regarding strings you should learn is the strlen() efun. This efun returns the length in characters of an LPC string, and is thus the string equivalent to sizeof() for arrays. Just from the behaviour of this efun, you can see that the driver treats a string as if it were made up of smaller elements. In this chapter, you will learn how to deal with strings on a more basic level, as characters and sub strings.

5.2 Strings as Character Arrays

You can do nearly anything with strings that you can do with arrays, except assign values on a character basis. At the most basic, you can actually refer to character constants by enclosing them in '' (single quotes). 'a' and "a" are therefore very different things in LPC. 'a' represents a character which cannot be used in assignment statements or any other operations except comparison evaluations. "a" on the other hand is a string made up of a single character. You can add and subtract other strings to it and assign it as a value to a variable.

With string variables, you can access the individual characters to run comparisons against character constants using exactly the same syntax that is used with arrays. In other words, the statement:

```
    if(str[2] == 'a')
```

is a valid LPC statement comparing the second character in the str string to the character 'a'. You have to be very careful that you are not comparing elements of arrays to characters, nor are you comparing characters of strings to strings.

LPC also allows you to access several characters together using LPC's range operator ..:

```
    if(str[0..1] == "ab")
```

In other words, you can look for the string which is formed by the characters 0 through 1 in the string str. As with arrays, you must be careful when using indexing or range operators so that you do not try to reference an index number larger than the last index. Doing so will result in an error.

Now you can see a couple of similarities between strings and arrays:
1) You may index on both to access the values of individual elements.
        a) The individual elements of strings are characters
        b) The individual elements of arrays match the data type of the array.
2) You may operate on a range of values
        a) Ex: "abcdef"[1..3] is the string "bcd"
        b) Ex: ({ 1, 2, 3, 4, 5 })[1..3] is the int array ({ 2, 3, 4 })

And of course, you should always keep in mind the fundamental difference: a string is not made up of a more fundamental LPC data type.

In other words, you may not act on the individual characters by assigning them values.

5.3 The Efun sscanf()
You cannot do any decent string handling in LPC without using sscanf(). Without it, you are left trying to play with the full strings passed by command statements to the command functions. In other words, you could not handle a command like: "give sword to leo", since you would have no way of separating "sword to leo" into its constituent parts. Commands such as these therefore use this efun in order to use commands with multiple arguments or to make commands more "English-like".

Most people find the manual entries for sscanf() to be rather difficult reading. The function does not lend itself well to the format used by manual entries. As I said above, the function is used to take a string and break it into usable parts. Technically it is supposed to take a string and scan it into one or more variables of varying types. Take the example above:

```
int give(string str) {
    string what, whom;

    if(!str) return notify_fail("Give what to whom?\n");
    if(sscanf(str, "%s to %s", what, whom) != 2)
      return notify_fail("Give what to whom?\n");
    ... rest of give code ...
}
```

The efun sscanf() takes three or more arguments. The first argument is the string you want scanned. The second argument is called a control string. The control string is a model which demonstrates in what form the original string is written, and how it should be divided up. The rest of the arguments are variables to which you will assign values based upon the control string.

The control string is made up of three different types of elements: 1) constants, 2) variable arguments to be scanned, and 3) variable arguments to be discarded. You must have as many of the variable arguments in sscanf() as you have elements of type 2 in your control string. In the above example, the control string was "%s to %s", which is a three element control string made up of one constant part (" to "), and two variable arguments to be scanned ("%s"). There were no variables to be discarded.

The control string basically indicates that the function should find the string " to " in the string str. Whatever comes before that constant will be placed into the first variable argument as a string. The same thing will happen to whatever comes after the constant.

Variable elements are noted by a "%" sign followed by a code for decoding them. If the variable element is to be discarded, the "%" sign is followed by the "*" as well as the code for decoding the variable. Common codes for variable element decoding are "s" for strings and "d" for integers. In addition, your mudlib may support other conversion codes, such as "f" for float. So in the two examples above, the "%s" in the control string indicates that whatever lies in the original string in the corresponding place will be scanned into a new variable as a string.

A simple exercise. How would you turn the string "145" into an integer?

Answer:
```
int x;
sscanf("145", "%d", x);
```

After the sscanf() function, x will equal the integer 145.

Whenever you scan a string against a control string, the function searches the original string for the first instance of the first constant in the original string. For example, if your string is "magic attack 100" and you have the following:
```
int improve(string str) {
    string skill;
    int x;

    if(sscanf(str, "%s %d", skill, x) != 2) return 0;
    ...
}
```
you would find that you have come up with the wrong return value for sscanf() (more on the return values later). The control string, "%s %d", is made up of to variables to be scanned and one constant. The constant is " ". So the function searches the original string for the first instance of " ", placing whatever comes before the " " into skill, and trying to place whatever comes after the " " into x. This separates "magic attack 100" into the components "magic" and "attack 100". The function, however, cannot make heads or tales of "attack 100" as an integer, so it returns 1, meaning that 1 variable value was successfully scanned ("magic" into skill).

Perhaps you guessed from the above examples, but the efun sscanf() returns an int, which is the number of variables into which values from the original string were successfully scanned. Some examples with return values for you to examine:

```
sscanf("swo  rd descartes", "%s to %s", str1, str2)           return: 0
sscanf("swo  rd descartes", "%s %s", str1, str2)              return: 2
sscanf("200 gold to descartes", "%d %s to %s", x, str1, str2) return: 3
sscanf("200 gold to descartes", "%d %*s to %s", x, str1)      return: 2
```
where x is an int and str1 and str2 are string

5.4 Summary
LPC strings can be thought of as arrays of characters, yet always keeping in mind that LPC does not have the character data type (with most, but not all drivers). Since the character is not a true LPC data type, you cannot act upon individual characters in an LPC string in the same manner you would act upon different data types. Noticing the intimate relationship between strings and arrays nevertheless makes it easier to understand such concepts as the range operator and indexing on strings.

There are efuns other than sscanf() which involve advanced string handling, however, they are not needed nearly as often. You should check on your mud for man or help files on the efuns: explode(), implode(), replace_string(), sprintf(). All of these are very valuable tools, especially if you intend to do coding at the mudlib level.

Chapter 6: Intermediate Inheritance

6.1 Basics of Inheritance

In the textbook LPC Basics, you learned how it is the mudlib maintains consistency amoung mud objects through inheritance. Inheritance allows the mud administrators to code the basic functions and such that all mudlib objects, or all mudlib objects of a certain type must have so that you can concentrate on creating the functions which make these objects different. When you build a room, or a weapon, or a monster, you are taking a set of functions already written for you and inheriting them into your object. In this way, all objects on the mud can count on other objects to behave in a certain manner. For instance, player objects can rely on the fact that all room objects will have a function in them called query_long() which describes the room. Inheritance thus keeps you from having to worry about what the function query_long() should look like.

Naturally, this textbook tries to go beyond this fundamental knowledge of inheritance to give the coder a better undertstanding of how inheritance works in LPC programming. Without getting into detail that the advanced domain coder/beginner mudlib coder simply does not yet need, this chapter will try to explain exactly what happens when you inherit an object.

6.2 Cloning and Inheritance

Whenever a file is referenced for the first time as an object (as opposed to reading the contents of the file), the game tries to load the file into memory and create an object. If the object is successfully loaded into memory, it becomes as master copy. Master copies of objects may be cloned but not used as actual game objects. The master copy is used to support any clone objects in the game.

The master copy is the source of one of the controversies of mud LPC coding, that is whether to clone or inherit. With rooms, there is no question of what you wish to do, since there should only be one instance of each room object in the game. So you generally use inheritance in creating rooms. Many mud administrators, including myself, however encourage creators to clone the standard monster object and configure it from inside room objects instead of keeping monsters in separate files which inherit the standard monster object.

As I stated above, each time a file is referenced to create an object, a master copy is loaded into memory. When you do something like:

```
void reset() {
    object ob;
    ob = new("/std/monster");
      /* clone_object("/std/monster") some places */
    ob->set_name("foo monster");
    ...  rest of monster config code followed by moving
it to the room ...
}
```

the driver searches to see if their is a master object called "/std/monster". If not, it creates one. If it does exist, or after it has been created, the driver then creates a clone object called "/std/monster#". If this is the first time "/std/monster" is being referenced, in effect, two objects are being created: the master object and the cloned instance.

On the other hand, let's say you did all your configuring in the create() of a special monster file which inherits "/std/monster". Instead of cloning the standard monster object from your room, you clone your monster file. If the standard monster has not been loaded, it gets loaded since your monster inherits it. In addition, a master copy of your file gets loaded into memory. Finally, a clone of your monster is created and moved into the room, for a total of three objects added to the game. Note that you cannot make use of the master copy easily to get around this. If, for example, you were to do:

```
    "/wizards/descartes/my_monster"->move(this_object());
```

instead of

```
    new("/wizards/descartes/my_monster")->move(this_object());
```

you would not be able to modify the file "my_monster.c" and update it, since the update command destroys the current master version of an object. On some mudlibs it also loads the new version into memory. Imagine the look on a player's face when their monster disappears in mid-combat cause you updated the file!

Cloning is therefore a useful too when you plan on doing just that- cloning. If you are doing nothing special to a monster which cannot be done through a few call others, then you will save the mud from getting loaded with useless master copies. Inheritance, however, is useful if you plan to add functionality to an object (write your own functions) or if you have a single configuration that gets used over and over again (you have an army of orc guards all the same, so you write a special orc file and clone it).

6.3 Inside Inheritance
When objects A and B inherit object C, all three objects have their own set of data sharing one set of function definitions from object C. In addition, A and B will have separate functions definitions which were entered separately into their code. For the sake of example throughout the rest of the chapter, we will use the following code. Do not be disturbed if, at this point, some of the code makes no sense:

```
OBJECT C
private string name, cap_name, short, long;
private int setup;

void set_name(string str)
nomask string query_name();
private int query_setup();
static void unsetup();
void set_short(string str);
string query_short();
void set_long(string str);
string query_long();


void set_name(string str) {
    if(!query_setup()) {
        name = str;
    setup = 1;
}

nomask string query_name() { return name; }

private query_setup() { return setup; }

static void unsetup() { setup = 0; }

string query_cap_name() {
    return (name ? capitalize(name) : ""); }
}

void set_short(string str) { short = str; }

string query_short() { return short; }

void set_long(string str) { long = str; }

string query_long() { return str; }

void create() { seteuid(getuid()); }

OBJECT B
inherit "/std/objectc";

private int wc;

void set_wc(int wc);
int query_wc();
int wieldweapon(string str);

void create() { ::create(); }

void init() {
    if(environment(this_object()) == this_player())
      add_action("wieldweapon", "wield");
}

void set_wc(int x) { wc = x; }

int query_wc() { return wc; }

int wieldweapon(string str) {
    ... code for wielding the weapon ...
}

OBJECT A
inherit "/std/objectc";

int ghost;

void create() { ::create(); }

void change_name(string str) {
    if(!((int)this_object()->is_player())) unsetup();
    set_name(str);
}
```

```
string query_cap_name() {
    if(ghost) return "A ghost";
    else return ::query_cap_name();
}
```

As you can see, object C is inherited both by object A and object B. Object C is a representation of a much oversimplified base object, with B being an equally oversimplified weapon and A being an equally simplified living object. Only one copy of each function is retained in memory, even though we have here three objects using the functions. There are of course, three instances of the variables from Object C in memory, with one instance of the variables of Object A and Object B in memory. Each object thus gets its own data.

6.4 Function and Variable Labels
Notice that many of the functions above are proceeded with labels which have not yet appeared in either this text or the beginner text, the labels static, private, and nomask. These labels define special priveledges which an object may have to its data and member functions. Functions you have used up to this point have the default label public. This is default to such a degree, some drivers do not support the labeling.

A public variable is available to any object down the inheritance tree from the object in which the variable is declared. Public variables in object C may be accessed by both objects A and B. Similarly, public functions may be called by any object down the inheritance tree from the object in which they are declared.

The opposite of public is of course private. A private variable or function may only be referenced from inside the object which declares it. If object A or B tried to make any reference to any of the variables in object C, an error would result, since the variables are said to be out of scope, or not available to inheriting classes due to their private labels. Functions, however, provide a unique challenge which variables do not. External objects in LPC have the ability to call functions in other objects through call others. The private label does not protect against call others.

To protect against call others, functions use the label static. A function which is static may only be called from inside the complete object or from the game driver. By complete object, I mean object A can call static functions in the object C it inherits. The static only protects against external call others. In addition, this_object()->foo() is considered an internal call as far as the static label goes.

Since variables cannot be referenced externally, there is no need for an equivalent label for them. Somewhere along the line, someone decided to muddy up the waters and use the static label with variables to have a completely separate meaning. What is even more maddening is that this label has nothing to do with what it means in the C programming language. A static variable is simply a variable that does not get saved to file through the efun save_object() and does not get restored through restore_object(). Go figure.

In general, it is good practice to have private variables with public functions, using query_*() functions to access the values of inherited variables, and set_*(), add_*(), and other such functions to change those values. In realm coding this is not something one really has to worry a lot about. As a matter of fact, in realm coding you do not have to know much of anything which is in this chapter. To be come a really good realm coder, however, you have to be able to read the mudlib code. And mudlib code is full of these labels. So you should work around with these labels until you can read code and understand why it is written that way and what it means to objects which inherit the code.

The final label is nomask, and it deals with a property of inheritance which allows you to rewrite functions which have already been defined. For example, you can see above that object A rewrote the function query_cap_name(). A rewrite of function is called overriding the function. The most common override of a function would be in a case like this, where a condition peculiar to our object (object A) needs to happen on a call ot the function under certain circumstances. Putting test code into object C just so object A can be a ghost is plain silly. So instead, we override query_cap_name() in object A, testing to see if the object is a ghost. If so, we change what happens when another object queries for the cap name. If it is not a ghost, then we want the regular object behaviour to happen. We therefore use the scope resolution operator (::) to call the inherited version of the query_cap_name() function and return its value.

A nomask function is one which cannot be overridden either through inheritance or through shadowing. Shadowing is a sort of backwards inheritance which will be detailed in the advanced LPC textbook. In the example above, neither object A nor object B (nor any other object for that matter) can override query_name(). Since we want to use query_name() as a unique identifier of objects, we don't want people faking us through shadowing or inheritance. The function therefore gets the nomask label.

6.5 Summary

Through inheritance, a coder may make user of functions defined in other objects in order to reduce the tedium of producing masses of similar objects and to increase the consistency of object behaviour across mudlib objects. LPC inheritance allows objects maximum priveledges in defining how their data can be accessed by external objects as well as objects inheriting them. This data security is maintained through the keywords, nomask, private, and static.

In addition, a coder is able to change the functionality of non-protected functions by overriding them. Even in the process of overriding a function, however, an object may access the original function through the scope resolution operator.

Chapter 7: Debugging

7.1 Types of Errors

By now, you have likely run into errors here, there, and everywhere. In general, there are three sorts of errors you might see: compile time errors, run time errors, and malfunctioning code. On most muds you will find a personal file where your compile time errors are logged. For the most part, this file can be found either in your home directory as the file named "log" or ".log", or somewhere in the directory "/log" as a file with your name.. In addition, muds tend to keep a log of run time errors which occur while the mud is up. Again, this is generally found in "/log". On MudOS muds it is called "debug.log". On other muds it may be called something different like "lpmud.log". Ask your administrators where compile time and run time errors are each logged if you do not already know.

Compile time errors are errors which occur when the driver tries to load an object into memory. If, when the driver is trying to load an object into memory, it encounters things which it simply does not understand with respect to what you wrote, it will fail to load it into memory and log why it could not load the object into your personal error log. The most common compile time errors are typos, missing or extra (), {}. [], or "", and failure to declare properly functions and variables used by the object.

Run time errors occur when something wrong happens to an object in memory while it is executing a statement. For example, the driver cannot tell whether the statement "x/y" will be valid in all circumstances. In fact, it is a valid LPC expression. Yet, if the value of y is 0, then a run time error will occur since you cannot divide by 0. When the driver runs across an error during the execution of a function, it aborts execution of the function and logs an error to the game's run time error log. It will also show the error to this_player(), if defined, if the player is a creator, or it will show "What?" to players. Most common causes for run time errors are bad values and trying to perform operations with data types for which those operations are not defined.

The most insideous type of error, however, is plain malfunctioning code. These errors do not log, since the driver never really realizes that anything is wrong. In short, this error happens when you think the code says one thing, but in fact it says another thing. People too often encounter this bug and automatically insist that it must be a mudlib or driver bug. Everyone makes all types of errors though, and more often than not when code is not functioning the way you should, it will be because you misread it.

7.2 Debugging Compile Time Errors

Compile time errors are certainly the most common and simplest bugs to debug. New coders often get frustrated by them due to the obscure nature of some error messages. Nevertheless, once a person becomes used to the error messages generated by their driver, debugging compile time errors becomes utterly routine.

In your error log, the driver will tell you the type of error and on which line it finally noticed there was an error. Note that this is not on which line the actual error necessarily exists. The most common compile time error, besides the typo, is the missing or superfluous parentheses, brackets, braces, or quotes. Yet this error is the one that most baffles new coders, since the driver will not notice the missing or extra piece until well after the original. Take for example the following code:

```
1 int test(string str) {
2    int x;
3    for(x =0; x<10; x++)
4        write(x+"\n");
5    }
```

```
6    write("Done.\n");
7 }
```

Depending on what you intended, the actual error here is either at line 3 (meaning you are missing a {) or at line 5 (meaing you have an extra }). Nevertheless, the driver will report that it found an error when it gets to line 6. The actual driver message may vary from driver to driver, but no matter which driver, you will see an error on line 6, since the } in line 5 is interpreted as ending the function test(). At line 6, the driver sees that you have a write() sitting outside any function definition, and thus reports an error. Generally, the driver will also go on to report that it found an error at line 7 in the form of an extra }.

The secret to debugging these is coding style. Having closing } match up vertically with the clauses they close out helps you see where you are missing them when you are debugging code. Similarly, when using multiple sets of parentheses, space out different groups like this:

```
    if( (x=sizeof(who=users()) > ( (y+z)/(a-b) + (-(random(7))) ) )
```

As you can see, the parentheses for the for() statement, are spaced out from the rest of the statement. In addition, individual sub-groups are spaced so they can easily be sorted out in the event of an error.

Once you have a coding style which aids in picking these out, you learn which error messages tend to indicate this sort of error. When debugging this sort of error, you then view a section of code before and after the line in question. In most all cases, you will catch the bug right off.

Another common compile time error is where the driver reports an unknown identifier. Generally, typos and failure to declare variables causes this sort of error. Fortunately, the error log will almost always tell you exactly where the error is. So when debugging it, enter the editor and find the line in question. If the problem is with a variable and is not a typo, make sure you declared it properly. On the other hand, if it is a typo, simply fix it!

One thing to beware of, however, is that this error will sometimes be reported in conjunction with a missing parentheses, brackets, or braces type error. In these situations, your problem with an unknown identifier is often bogus. The driver misreads the way the {} or whatever are setup, and thus gets variable declarations confused. Therefore make sure all other compile time errors are corrected before bothering with these types of errors.

In the same class with the above error, is the general syntax error. The driver generates this error when it simply fails to understand what you said. Again, this is often caused by typos, but can also be caused by not properly understanding the syntax of a certain feature like writing a for() statement: for(x=0, x<10, x++). If you get an error like this which is not a syntax error, try reviewing the syntax of the statement in which the error is occurring.

7.3 Debugging Run Time Errors
Run time errors are much more complex than their compile time counterparts. Fortunately these errors do get logged, though many creators do not realise or they do not know where to look. The error log for run time errors are also generally much more detailed than compile

time errors, meaning that you can trace the history of the execution train from where it started to where it went wrong. You therefore can setup debugging traps using precompiler statements much easier using these logs. Run time errors, however, tend to result from using more complex codign techniques than beginners tend to use, which means you are left with errors which are generally more complicated than simple compile time errors.

Run time errors almost always result from misusing LPC data types. Most commonly, trying to do call others using object variables which are NULL, indexing on mapping, array, or string variables which are NULL, or passing bad arguments to functions. We will look at a real run time error log from Nightmare:

```
Bad argument 1 to explode()
program: bin/system/_grep.c, object: bin/system/_grep
line 32
'       cmd_hook' in '        std/living.c' ('
std/user#4002')line 83
'       cmd_grep' in '  bin/system/_grep.c' ('
bin/system/_grep')line 32
Bad argument 2 to message()
program: adm/obj/simul_efun.c, object: adm/obj/simul_efun
line 34
'       cmd_hook' in '        std/living.c' ('
std/user#4957')line 83
'       cmd_look' in '  bin/mortal/_look.c' ('
bin/mortal/_look')line 23
' examine_object' in '  bin/mortal/_look.c' ('
bin/mortal/_look')line 78
'          write' in 'adm/obj/simul_efun.c' ('
adm/obj/simul_efun')line 34
Bad argument 1 to call_other()
program: bin/system/_clone.c, object: bin/system/_clone
line 25
'       cmd_hook' in '        std/living.c' ('
std/user#3734')line 83
'      cmd_clone' in ' bin/system/_clone.c' ('
bin/system/_clone')line 25
Illegal index
program: std/monster.c, object:
wizards/zaknaifen/spy#7205 line 76
'     heart_beat' in '       std/monster.c'
('wizards/zaknaifen/spy#7205')line
76
```

All of the errors, except the last one, involve passing a bad argument to a function. The first bug, involves passing a bad first arument to the efun explode(). This efun expects a string as its first argment. In debugging these kinds of errors, we would therefore go to line 32 in /bin/system/_grep.c and check to see what the data type of the first argument being passed in fact is. In this particular case, the value being passed should be a string.

If for some reason I has actually passed something else, I would be done debugging at that point and fix it simply by making sure that I was passing a string. This situation is more complex. I now need to trace

the actual values contained by the variable being passed to explode, so
that I can see what it is the explode() efun sees that it is being passed.

The line is question is this:

```
 borg[files[i]] = regexp(explode(read_file(files[i]), "\n"), exp);
```

where files is an array for strings, i is an integer, and borg is a mapping.
So clearly we need to find out what the value of read_file(files[i]) is.
Well, this efun returns a string unless the file in question does not exist,
the object in question does not have read access to the file in question, or
the file in question is an empty file, in which cases the function will
return NULL. Clearly, our problem is that one of these events must
have happened. In order to see which, we need to look at files[i].

Examining the code, the files array gets its value through the get_dir()
efun. This returns all the files in a directory if the object has read access
to the directory. Therefore the problem is neither lack of access or non-
existent files. The file which caused this error then must have been an
empty file. And, in fact, that is exactly what caused this error. To
debug that, we would pass files through the filter_array() efun and make
sure that only files with a file size greater than 0 were allowed into the
array.

The key to debugging a run time error is therefore knowing exactly what
the values of all variables in question are at the exact moment where the
bug created. When reading your run time log, be careful to separate the
object from the file in which the bug occurred. For example, the
indexing error above came about in the object /wizards/zaknaifen/spy,
but the error occured while running a function in /std/monster.c, which
the object inherited.

7.4 Malfunctioning Code

The nastiest problem to deal with is when your code does not behave the
way you intended it to behave. The object loads fine, and it produces no
run time errors, but things simply do not happen the way they should.
Since the driver does not see a problem with this type of code, no logs
are produced. You therefore need to go through the code line by line
and figure out what is happening.

Step 1: Locate the last line of code you knew successfully executed
Step 2: Locate the first line of code where you know things are going
wrong
Step 3: Examine the flow of the code from the known successful point to
the first known unsuccessful point.

More often than not, these problems occurr when you are using if()
statements and not accounting for all possibilities. For example:

```
int cmd(string tmp) {
    if(stringp(tmp)) return do_a()
    else if(intp(tmp)) return do_b()
    return 1;
}
```

In this code, we find that it compiles and runs fine. Problem is nothing
happens when it is executed. We know for sure that the cmd() function
is getting executed, so we can start there. We also know that a value of
1 is in fact being returned, since we do not see "What?" when we enter

the command. Immediately, we can see that for some reason the variable tmp has a value other than string or int. As it turns out, we issued the command without parameters, so tmp was NULL and failed all tests.

The above example is rather simplistic, bordering on silly. Nevertheless, it gives you an idea of how to examine the flow of the code when debugging malfunctioning code. Other tools are available as well to help in debugging code. The most important tool is the use of the precompiler to debug code. With the code above, we have a clause checking for integers being passed to cmd(). When we type "cmd 10", we are expecting do_b() to execute. We need to see what the value of tmp is before we get into the loop:

```
#define DEBUG
int cmd(string tmp) {
#ifdef DEBUG
    write(tmp);
#endif
    if(stringp(tmp)) return do_a();
    else if(intp(tmp)) return do_b();
    else return 1;
}
```

We find out immediately upon issuing the command, that tmp has a value of "10". Looking back at the code, we slap ourselves silly, forgetting that we have to change command arguments to integers using sscanf() before evaluating them as integers.

7.5 Summary

The key to debugging any LPC problem is always being aware of what the values of your variables are at any given step in your code. LPC execution reduces on the simplest level to changes in variable values, so bad values are what causes bad things to happen once code has been loaded into memory. If you get errors about bad arguments to functions, more likely than not you are passing a NULL value to a function for that argument. This happens most often with objects, since people will do one of the following:

1) use a value that was set to an object that has since destructed
2) use the return value of this_player() when there is no this_player()
3) use the return value of this_object() just after this_object() was destructed

In addition, people will often run into errors involving illegal indexing or indexing on illegal types. Most often, this is because the mapping or array in question was not initialized, and therefore cannot be indexed. The key is to know exactly what the full value of the array or mapping should be at the point in question. In addition, watch for using index numbers larger than the size of given arrays

Finally, make use of the precompiler to temporarily throw out code, or introduce code which will show you the values of variables. The precompiler makes it easy to get rid of debugging code quickly once you are done. You can simply remove the DEBUG define when you are done.

Table of contents