LPC Basics

Written by Descartes of Borg
borg@hebron.connected.com
first edition: 23 april 1993
second edition: 25 may 1993

INTRODUCTION
This manual, how to use it, and its terms

I have seen a lot of requests lately on USENET for LPC manuals. In addition, the immortals on my mud have been telling how good the building documentation of Nightmare is, but that there was just no adequate explanation of the LPC programming language. So I decided to try my hand at writing a manual. Some things you should keep in mind.
LPC is a very easy programming language to learn, and it has real value in that place most of us know as the real world. I began playing muds in 1991, and in the space of a month created an unimpressive area and musician's guild on the original Bates College MUD called Orlith. After that, I moved to Los Angeles for a year and had no contact with mudding or computers. In June of 1992, I was back on the internet and a wizard of Igor. In September of 1992 I began coding the Nightmare mudlib for our use, and then later decided to distribute it due to there not being any mudlibs for MudOS at the time that a person could just throw up a running mud with (now, that of course is not the case :)).
So, I have been doing serious coding for less than a year. As a Philosophy major in a world of Computer Science majors, I just want to make clear that it is not at all required that you have ever done anything with your computer than log into a mud in order for you to really come to understand LPC coding. This manual makes the following assumptions: Someone has taught you basic UNIX commands like ls, cd, mkdir, mv, rm, etc. You know how to enter your mud's editor and write a file. No other assumptions are made. If you know C, you are handicapped in that LPC looks a lot like C, but it is not C. Your preconceptions about modular programming development will be a hinderence you will have to overcome. If you have never heard of the C programming language (like me in May of 1991), then you are only missing an understanding of the simple constructs of C like the flow of program execution and logical operators and such. So a C guru has no real advantage over you, since what they know from C which is applicable to LPC is easy to pick up. The stuff they know about C which makes them a guru is irrelevant to LPC.

The chapters of this manual are meant to be read in order. Starting with the introduction, going sequentially through the chapter numbers as

ordered in the contents file. Each chapter begins with a paragraph or two explaining what you should have come to understand by that point in your studies. After those introductory paragraphs, the chapter then begins to discuss its subject matter in nauseating detail. At the end of the chapter is a briefly worded summary of what you should understand from that chapter if I have been successful. Following that may or may not be some sidenotes relevant to the subject at hand, but not necessary to its understanding.

If at any time you get to a chapter intro, and you have read the preceeding chapters thoroughly and you do not understand what it says you should understand by that point, please mail me! Clearly, I have failed at that point and I need to know where it is I have gone wrong so I can revise it properly. Similarly, if you do not understand what the chapter summary says you should, please mail me. If your mumud is on the MudOS intermud system, mail descartes@nightmare. Otherwise mail borg@hebron.connected.com.

Some basic terms this manual uses:
driver-
This is the C program which is the game. It accepts incoming sockets (links to other computers), interprets LPC code defined by the mudlib, keeps mud objects in memory, makes periodic attempts to clean unused mud objects from memory, makes periodic calls to objects, and so on.

mudlib-
LPC code which defines the world in which you are in. The driver of itself is not a game. It is just a program which allows the creation of a multi-user environment. In some sense, the driver is like an LPC compiler, and the mudlib is like a compiler's library (a very loose analogy). The mudlib defines basic objects which will likely be used over and over again by people creating in the mud world. Examples of such objects are /std/room (or /room/room), /std/user.c (or /obj/player.c), and so on.

area or castle:
Specific creator coded objects which often use a feature of LPC called inheritance to make use of the properties of basic mudlib objects and turn them into specific objects to be used by players in the game

object:
a room, a weapon, a monster, a player, a bag, etc. More importantly, every individual file with a .c extension is an object. Objects are used in different ways. Objects like /std/living.c are inherited by objects like monster.c and user.c. Others are cloned, which means a duplicate of that code is loaded into memory. And still others are simply loaded into memory to be referenced by other objects.

native and compat:
these two terms refer to two popular flavours of drivers. Native mode mudlibs make use of on the design of LPMud driver 3.0 and later. You may have a 3.0 driver however, but have a 2.4.5 style mudlib. This is what is meant by compat mode. Mudlibs which are native mode are any for MudOS, CD, and LPMud mudlibs that
are listed as native. Compat mudlibs are any LPMud mudlib before 3.0 and those which are 3.* compat mudlibs. I believe Amylaar's is compat.

CHAPTER 1: Introduction to the Coding Environment

1.1 UNIX file structure

LPMuds use basic UNIX commands and its file structure. If you know UNIX commands already, then note (with a few exceptions) options are not available to the commands. Like DOS, UNIX is heirarchical. The root directory of which all directories are sub-directories is called root(/). And from those sub-directories you may have further sub-directories. A directory may be referred to in two different ways: 1) by its full name, or absolute name, or 2) by its relative name. Absolute name refers to the directory's full path starting from / winding down the directory tree until you name the directory in question. For example:

```
/players/descartes/obj/monster
```

refers to the directory monster which is a sub-directory of obj which is a sub-directory of descartes which is a sub-directory of players which is a sudirectory of /.

The relative name refers to the name relative to another directory. The above example is called monster relative to /players/descartes/obj, but it is also called obj/monster relative to /players/descartes, descartes/obj/monster relative to /players, and finally players/descartes/obj/monster relative to /. You can tell the difference between absolute names and relative names because absolute names always start with /. In order to know exactly which directory is being named by a relative name, you naturally must know what directory it is relative to.

A directory contains sub-directories and files. LPMuds only use text files inside the mudlib. Like directories, files have both absolute and relative names. The most basic relative name is often referred to as the file name, with the rest of the absolute name being referred to as the path. So, for the file: /players/descartes/castle.c, castle.c is the file name, and /players/descartes is the path.

On some muds, a file with a file name beginning with a . (like .plan) is not visible when you list files with the regular file listing command.

1.2 UNIX Commands

Along with the UNIX file structure, LPMuds use many UNIX commands. Typical UNIX commands on most muds are:
pwd, cd, ls, rm, mv, cp, mkdir, rmdir, more, head, cat, ed
If you have never before seen UNIX commands, you probably are thinking this is all nonsense. Well, it is, but you got to use them. Before getting into what they mean though, first a discussion of current directory. If you know DOS, then you know what a current working directory is. At any given point, you are considered to be "in" some directory. This means that any relative file or directory names you give in UNIX commands are relative to that directory. For example, if my current directory is /players/descartes and I type "ed castle.c" (ed is the command to edit), then it assumes I mean the file /players/descartes/castle.c

pwd: shows you your current working directory
cd: changes your current working directory. You may give either relative or absolute path names. With no arguments, it changes to your home

```
    directory.
ls: lists all files in the directory named.  If no directory is named,
    it lists the files of the current working directory
rm: deletes the file named
mv: renames the file named
cp: copies the file named
mkdir: makes a new directory
rmdir: deletes a directory.  All files must have been first removed.
more: pages the file named so that the file appears on your screen one
    page at a time.
cat: shows the whole file to you at once
head: shows you the first several lines of a file
tail: shows you the last several lines of a file
ed: allows you to edit a file using the mud editor
```

1.3 Chapter Summary

UNIX uses a heirarchical file structure with the root of the tree being named /. Other directories branch off from that root directory and in turn have their own sub-directories. All directories may contain directories and files. Directories and files are referred to either by their absolute name, which always begins with /, or by their relative name which gives the file's name relative to a particular directory. In order to get around in the UNIX files structure, you have the typical UNIX commands for listing files, your current directory, etc. On your mud, all of the above commands should have detailed help commands to help you explore exactly what they do. In addition, there should be a very detailed file on your mud's editor. If you are unfamiliar with ed, you should go over this convoluted file.

CHAPTER 2: The LPC Program

2.1 About programs

The title of this chapter of the textbook is actually poorly named, since one does not write programs in LPC. An LPC coder instead writes *objects*. What is the difference? Well, for our purposes now, the difference is in the way the file is executed. When you "run" a program, execution begins at a definite place in the program. In other words, there is a place in all programs that is noted as the beginning where program execution starts. In addition, programs have definite end points, so that when execution reaches that point, the execution of the program terminates. So, in short, execution of a program runs from a definite beginning point through to a definite end point. This is not so with LPC objects.

With muds, LPC objects are simply distinct parts of the C program which is running the game (the driver). In other words, execution of the mud program begins and ends in the driver. But the driver in fact does very little in the way of creating the world you know when you play a mud. Instead, the driver relies heavily on the code created in LPC, executing lines of the objects in the mud as needed. LPC objects thus have no place that is necessarily the beginning point, nor do they have a definite ending point.

Like other programming languages, an LPC "program" may be made up of one or more files. For an LPC object to get executed, it simple needs to be loaded into the driver's memory. The driver will call lines

from the object as it needs according to a structure which will be defined throughout this textbook. The important thing you need to understand at this point is that there is no "beginning" to an LPC object in terms of execution, and there is no "end".

2.2 Driver-mudlib interaction

As I have mentioned earlier, the driver is the C program that runs on the host machine. It connects you into the game and processes LPC code. Note that this is one theory of mud programming, and not necessarily better than others. It could be that the entire game is written in C. Such a game would be much faster, but it would be less flexible in that wizards could not add things to the game while it was running. This is the theory behind DikuMUDs. Instead, LPMUDs run on the theory that the driver should in no define the nature of the game, that the nature of the game is to be decided by the individuals involved, and that you should be able to add to the game *as it is being played*. This is why LPMUDs make use of the LPC programming language. It allows you to define the nature of the game in LPC for the driver to read and execute as needed. It is also a much simpler language to understand than C, thus making the process of world creation open to a greater number of people.

Once you have written a file in LPC (assuming it is corrent LPC), it justs sits there on the host machine's hard drive until something in the game makes reference to it. When something in the game finally does make reference to the object, a copy of the file is loaded into memory and a special *function* of that object is called in order to initialize the values of the variables in the object. Now, do not be concerned if that last sentence went right over your head, since someone brand new to programming would not know what the hell a function or a variable is. The important thing to understand right now is that a copy of the object file is taken by the driver from the machine's hard drive and stored into memory (since it is a copy, multiple versions of that object may exist). You will later understand what a function is, what a variable is, and exactly how it is something in the game made reference to your object.

2.3 Loading an object into memory

Although there is no particular place in an object code that must exist in order for the driver to begin executing it, there is a place for which the driver will search in order to initialize the object. On compat drivers, it is the function called reset(). On native muds it is the function called create().

LPC objects are made up of variables (values which can change) and functions which are used to manipulate those variables. Functions manipulate variables through the use of LPC grammatical structures, which include calling other functions, using externally defined functions (efuns), and basic LPC expressions and flow control mechanisms.

Does that sound convoluted? First lets start with a variable. A variable might be something like: level. It can "vary" from sitation to situation in value, and different things use the value of the player's level to make different things happen. For instance, if you are a level 19 player, the value of the variable level will be 19. Now if your mud is on the old LPMud 2.4.5 system where levels 1-19 are

players and 20+ are wizards, things can ask for your level value to see if you can perform wizard type actions. Basically, each object in LPC is a pile of variables with values which change over time. Things happen to these objects based on what values its variables hold. Often, then things that happen cause the variables to change.

So, whenever an object in LPC is referenced by another object currently in memory, the driver searches to see what places for values the object has (but they have no values yet). Once that is done, the driver calls a function in the object called reset() or create() (depending on your driver) which will set up the starting values for the object's variables. It is thus through *calls* to *functions* that variable values get manipulated.

But create() or reset() is NOT the starting place of LPC code, although it is where most LPC code execution does begin. The fact is, those functions need not exist. If your object does just fine with its starting values all being NULL pointers (meaning, for our purposes here, 0), then you do not need a create() or reset() function. Thus the first bit of execution of the object's code may begin somewhere completely different.

Now we get to what this chapter is all about. The question: What consists a complete LPC object? Well, an LPC object is simply one or more functions grouped together manipulating 0 or more variables. The order in which functions are placed in an object relative to one another is irrelevant. In other words:

```
-----
void init() { add_action("smile", "smile"); }

void create() { return; }

int smile(string str) { return 0; }
-----
```

is exactly the same as:

```
-----
void create() { return; }

int smile(string str) { return 0; }

void init() { add_action("smile", "smile"); }
_____
```

Also important to note, the object containing only:

```
-----
void nonsense() {}
-----
```

is a valid, but trivial object, although it probably would not interact properly with other objects on your mud since such an object has no weight, is invisible, etc..

2.4 Chapter summary

LPC code has no beginning point or ending point, since LPC code is used to create objects to be used by the driver program rather than create individual programs. LPC objects consist of one or more functions whose order in the code is irrelevant, as well as of zero or more variables whose values are manipulated inside those functions. LPC objects simply sit on the host machine's hard driver until referenced by another object in the game (in other words, they do not really exist). Once the object is referenced, it is loaded into the machine's memory with empty values for the variables. The function reset() in compat muds or create() in native muds is called in that object if it exists to allow the variables to take on initial values. Other functions in the object are used by the driver and other objects in the game to allow interaction among objects and the manipulation of the LPC variables.

A note on reset() and create():
create() is only used by muds in native mode (see the textbook Introduction for more information on native mode vs. compat mode). It is only used to initialize newly referenced objects.

reset() is used by both muds in compat mode and native mode. In compat mode, reset() performs two functions. First, it is used to initialize newly referenced objects. In addition, however, compat mode muds use reset() to "reset" the object. In other words, return it to its initial state of affairs. This allows monsters to regenerate in a room and doors to start back in the shut position, etc.. Native mode muds use reset() to perform the second function (as its name implies).

So there are two important things which happen in LP style muds which cause the driver to make calls to functions in objects. The first is the creation of the object. At this time, the driver calls a function to initalize the values in the object. For compat mode muds, this is performed by the function named reset() (with an argument of 0, more on this later though). For muds running in native mode, this is performed by the function create().

The second is the returning of the room to some base state of affairs. This base set of affairs may or may not be different from the initial state of affairs, and certainly you would not want to take up time doing redundant things (like resetting variables that never change). Compat mode muds nevertheless use the same function that was used to create the object to reset it, that being reset(). Native mode muds, who use create() to create the room, instead use reset() to reset it. All is not lost in compat mode though, as there is a way to tell the difference between creation and resetting. For reset purposes, the driver passes either 1 or the reset number as an argument to reset() in compat mode. Now this is meaningless to you now, but just keep in mind that you can in fact tell the difference in compat mode. Also keep in mind that the argment in the creation use of reset is 0 and the argument in the reset use is a nonzero number.

CHAPTER 3: LPC Data Types

3.1 What you should know by now

LPC object are made up of zero or more variables manipulated by one or more functions. The order in which these functions appear in code is irrelevant. The driver uses the LPC code you write by loading copies of

it into memory whenever it is first referenced and additional copies through cloning. When each object is loaded into memory, all the variables initially point to no value. The reset() function in compat muds, and create() in native muds are used to give initial values to variables in objects. The function for creation is called immediately after the object is loaded into memory. However, if you are reading this textbook with no prior programming experience, you may not know what a function is or how it gets called. And even if you have programming experience, you may be wondering how the process of functions calling each other gets started in newly created objects. Before any of these questions get answered, however, you need to know more about what it is the functions are manipulating. You therefore should thouroughly come to know the concept behind LPC data types. Certainly the most boring subject in this manual, yet it is the most crucial, as 90% of all errors (excepting misplaced {} and ()) involve the improper usage of LPC data types. So bear through this important chapter, because it is my feeling that understanding this chapter alone can help you find coding much, much easier.

3.2 Communicating with the computer

You possibly already know that computers cannot understand the letters and numbers used by humans. Instead, the "language" spoken by computers consists of an "alphabet" of 0's and 1's. Certainly you know computers do not understand natural human languages. But in fact, they do not understand the computer languages we write for them either. Computer languages like BASIC, C, C++, Pascal, etc. are all intermediate languages. They allow you to structure your thoughts more coherently for translation into the 0's and 1's of the computer's languages.

There are two methods in which translation is done: compilation and interpretation. These simply are differences betweem when the programming language is translated into computer language. With compiled languages, the programmer writes the code then uses a program called a compiler to translate the program into the computer's language. This translation occurs before the program is run. With interpreted languages however, the process of translation occurs as the program is being run. Since the translation of the program is occurring during the time of the program's running in interpreted languages, interpreted languages make much slower programs than compiled languages.

The bottom line is, no matter what language you are writing in, at some point this has to be changed into 0's and 1's which can be understood by the computer. But the variables which you store in memory are not simply 0's and 1's. So you have to have a way in your programming languages of telling the computer whether or not the 0's and 1's should be treated as decimal numbers or characters or strings or anything else. You do this through the use of data types.

For example, say you have a variable which you call 'x' and you give it the decimal whole number value 65. In LPC you would do this through the statement:

```
-----
x = 65;
-----
```

You can later do things like:

```
_____
write(x+"\n");        /* \n is symbolically represents a carriage return */
y = x + 5;
-----
```

The first line allows you to send 65 and a carriage return to someone's screen.
The second line lets you set the value of y to 70.
The problem for the computer is that it does not know what '65' means when
you tell it x = 65;. What you think of 65, it might think of as:
00000000000000000000000001000001
But, also, to the computer, the letter 'A' is represented as:
00000000000000000000000001000001
So, whenever you instruct the computer write(x+"\n");, it must have some
way of knowing that you want to see '65' and not 'A'.

The computer can tell the difference between '65' and 'A' through the use
of data types. A data types simply says what type of data is being stored
by the memory location pointed to by a given variable. Thus, each LPC
variable has a variable type which guides conversions. In the example
given above, you would have had the following line somewhere in the
code *before* the lines shown above:

```
-----
int x;
-----
```

This one line tells the driver that whatever value x points to, it will
be used as the data type "int", which is short for integer, or whole
number. So you have a basic introduction into the reason why data types
exist. They exist so the driver can make sense of the 0's and 1's that
the computer is storing in memory.

3.3 The data types of LPC
All LPMud drivers have the following data types:

void, status, int, string, object, int *, string *, object *, mixed *

Many drivers, but not all have the following important data types which
are important to discuss:

float, mapping, float *, mapping *

And there are a few drivers with the following rarely used data types
which are not important to discuss:

function, enum, struct, char

3.4 Simple data types
This introductory textbook will deal with the data types void, status,
int, float, string, object, mand mixed. You can find out about the
more complex data types like mappings and arrays in the intermediate
textbook. This chapter deals with the two simplest data types (from the
point of view of the LPC coder), int and string.

An int is any whole number. Thus 1, 42, -17, 0, -10000023 are all type int.
A string is one or more alphanumeric characters. Thus "a", "we are borg",

"42", "This is a string" are all strings. Note that strings are always enclosed in "" to allow the driver to distinguish between the int 42 and the string "42" as well as to distinguish between variable names (like x) and strings by the same names (like "x").

When you use a variable in code, you must first let the driver know what type of data to which that variable points. This process is called *declaration*. You do this at the beginning of the function or at the beginning of the object code (outside of functions before all functions which use it). This is done by placing the name of the data type before the name of the variable like in the following example:

```
-----
void add_two_and_two() {
    int x;
    int y;

    x = 2;
    y = x + x;
}
-----
```

Now, this is a complete function. The name of the function is add_two_and_two(). The function begins with the declaration of an int variable named x followed by the declaration of an in variable named y. So now, at this point, the driver now has two variables which point to NULL values, and it expects what ever values end up there to be of type int.

A note about the data types void and status:
Void is a trivial data type which points to nothing. It is not used with respect to variables, but instead with respect to functions. You will come to understand this better later. For now, you need only understand that it points to no value.

The data type status is a boolean data type. That is, it can only have 1 or 0 as a value. This is often referred to as being true or false.

3.5 Chapter summary
For variables, the driver needs to know how the 0's and 1's the computer stores in memory get converted into the forms in which you intend them to be used. The simplest LPC data types are void, status, int, and string. You do not user variables of type void, but the data type does come into play with respect to functions. In addition to being used for translation from one form to the next, data types are used in determining what rules the driver uses for such operations as +, -, etc. For example, in the expression 5+5, the driver knows to add the values of 5 and 5 together to make 10. With strings however, the rules for int addition make no sense. So instead, with "a"+"b", it appends "b" to the string "a" so that the final string is "ab". Errors can thus result if you mistakenly try to add "5"+5. Since int addition makes no sense with strings, the driver will convert the second 5 to "5" and use string addition. The final result would be "55". If you were looking for 10, you would therefore have ended up with erroneous code. Keep in mind, however, that in most instances, the driver will not do something so useful as coming up with "55". It comes up with "55" cause it has a rule for adding a string to an int, namely to treat the int as a string. In most cases, if you

use a data type for which an operation or function is not defined
(like if you tried to divide "this is" by "nonsense", "this is"/"nonsense"),
the driver will barf and report an error to you.

CHAPTER 4: Functions

4.1 Review
By this point, you should be aware that LPC objects consist of functions
which manipulate variables. The functions manipulate variables when they
are executed, and they get executed through *calls* to those functions.
The order in which the functions are placed in a file does not matter.
Inside a function, the variables get manipulated. They are stored in
computer memory and used by the computer as 0's and 1's which
get translated to and from useable output and input through a device
called data typing. String data types tell the driver that the
data should appear to you and come from you in the form of alphanumeric
characters. Variables of type int are represented to you as whole
number values. Type status is represented to you as either 1 or 0.
And finally type void has no value to you or the machine, and is not
really used with variable data types.

4.2 What is a function?
Like math functions, LPC functions take input and return output.
Languages like Pascal distinguish between the concept of proceedure abd
the concept of function. LPC does not, however, it is useful to
understand this distinction. What Pascal calls a proceedure, LPC
calls a function of type void. In other words, a proceedure, or function
of type void returns no output. What Pascal calls a function differs
in that it does return output. In LPC, the most trivial, correct
function is:

```
-----
void do_nothing() { }
-----
```

This function accepts no input, performs no instructions, and returns no
value.

There are three parts to every properly written LPC function:
1) The declaration
2) The definition
3) The call

Like with variables, functions must be declared. This will allow the
driver to know 1) what type of data the function is returning as output,
and 2) how many input(s) and of what type those input(s) are. The
more common word for input is parameters.
A function declaration therefore consists of:
type name(parameter1, parameter2, ..., parameterN);
The declaration of a function called drink_water() which accepts a string as
input and an int as output would thus look like this:

```
-----
int drink_water(string str);
-----
```

where str is the name of the input as it will be used inside the function.

The function definition is the code which describes what the function actually does with the input sent to it.
The call is any place in other functions which invokes the execution of the function in question. For two functions write_vals() and add(), you thus might have the following bit of code:

```
-----
/* First, function declarations.  They usually appear at the beginning
   of object code.
*/
void write_vals();
int add(int x, int y);

/* Next, the definition of the function write_vals().  We assume that
   this function is going to be called from outside the object
*/
void write_vals() {
    int x;

    /*N Now we assign x the value of the output of add() through a call */
    x = add(2, 2);
    write(x+"\n");
}

/* Finally, the definition of add() */
int add(int x, int y) {
    return (x + y);
}
-----
```

Remember, it does not matter which function definition appears first in the code. This is because functions are not executed consecutively. Instead, functions are executed as called. The only requirement is that the declaration of a function appear before its definition and before the definition of any function which makes a call to it.

4.3 Efuns

Perhaps you have heard people refer to efuns. They are externally defined functions. Namely, they are defined by the mud driver. If you have played around at all with coding in LPC, you have probably found some expressions you were told to use like this_player(), write(), say(), this_object(), etc. look a lot like functions. That is because they are efuns. The value of efuns is that they are much faster than LPC functions, since they already exist in the binary form the computer understands.

In the function write_vals() above, two functions calls were made. The first was to the functions add(), which you declared and defined. The second call, however, was to a function called write(), and efun. The driver has already declared and defined this function for you. You needs only to make calls to it.

Efuns are created to hanldle common, every day function calls, to handle input/output to the internet sockets, and other matters difficult to be dealt with in LPC. They are written in C in the game driver and compiled along with the driver before the mud comes up, making them much faster in execution. But for your purposes, efun calls are just like calls

made to your functions. Still, it is important to know two things of any efun: 1) what return type does it have, and 2) what parameters of what types does it take.

Information on efuns such as input parameters and return types is often found in a directory called /doc/efun on your mud. I cannot detail efuns here, because efuns vary from driver to driver. However, you can often access this information using the commands "man" or "help" depending on your mudlib. For instance, the command "man write" would give you information on the write efun. But if all else fails, "more /doc/efun/write" should work.

By looking it up, you will find write is declared as follows:

```
-----
void write(string);
-----
```

This tells you an appropriate call to write expects no return value and passes a single parameter of type string.

4.4 Defining your own functions

Although ordering your functions within the file does not matter, ordering the code which defines a function is most important. Once a function has been called, function code is executed in the order it appears in the function definition. In write_vals() above, the instruction:

```
-----
x = add(2, 2);
-----
```

Must come before the write() efun call if you want to see the appropriate value of x used in write().

With respect to values returned by function, this is done through the "return" instruction followed by a value of the same data type as the function. In add() above, the instruction is "return (x+y);", where the value of (x+y) is the value returned to write_vals() and assigned to x. On a more general level, "return" halts the execution of a function and returns code execution to the function which called that function. In addition, it returns to the calling function the value of any expression that follows. To stop the execution of a function of type void out of order, use "return"; without any value following. Once again, remember, the data type of the value of any expression returned using "return" MUST be the same as the data type of the function itself.

4.5 Chapter Summary

The files which define LPC objects are made of of functions. Functions, in turn, are made up of three parts:

1) The declaration
2) The definition
3) The call

Function declarations generally appear at the top of the file before any defintions, although the requirement is that the declaration must appear before the function definition and before the definition of any function which calls it.

Function definitions may appear in the file in any order so long as they

come after their declaration. In addition, you may not define one function inside another function.
Function calls appear inside the definition of other functions where you want the code to begin execution of your function. They may also appear within the definition of the function itself, but this is not recommended for new coders, as it can easily lead to infinite loops.

The function definition consists of the following in this order:
1) function return type
2) function name
3) opening (followed by a parameter list and a closing)
4) an opening { instructing the driver that execution begins here
5) declarations of any variables to be used only in that function
6) instructions, expressions, and calls to other functions as needed
7) a closing } stating that the function code ends here and, if no "return" instruction has been given at this point (type void functions only), execution returns to the calling function as if a r"return" instruction was given

The trivial function would thus be:

```
-----
void do_nothing() {}
-----
```

since this function does not accept any input, perform any instructions, or return any output.

Any function which is not of type void MUST return a value of a data type matching the function's data type.

Each driver has a set of functions already defined for you called efuns These you need neither need to declare nor define since it has already been done for you. Furthermore, execution of these functions is faster than the execution of your functions since efuns are in the driver.
In addition, each mudlib has special functions like efuns in that they are already defined and declared for you, but different in that they are defined in the mudlib and in LPC. They are called simul_efuns, or simulated efuns. You can find out all about each of these as they are listed in the /doc/efun directory on most muds. In addition many muds have a command called "man" or a "help" command which allows you simply to call up the info files on them.

Note on style:
Some drivers may not require you to declare your functions, and some may not require you to specify the return type of the function in its definition. Regardless of this fact, you should never omit this information for the following reasons:
1) It is easier for other people (and you at later dates) to read your code and understand what is meant. This is particularly useful for debugging, where a large portion of errors (outside of misplaced parentheses and brackets) involve problems with data types (Ever gotten "Bad arg 1 to foo() line 32"?).
2) It is simply considered good coding form.

CHAPTER 5: The Basics of Inheritance

5.1 Review
You should now understand the basic workings of functions. You should be able to declare and call one. In addition, you should be able to recognize function definitions, although, if this is your first experience with LPC, it is unlikely that you will as yet be able to define your own functions. There functions form the basic building blocks of LPC objects. Code in them is executed when another function makes a call to them. In making a call, input is passed from the calling function into the execution of the called one. The called function then executes and returns a value of a certain data type to the calling function. Functions which return no value are of type void.

After examining your workroom code, it might look something like this (depending on the mudlib):

```
-----
inherit "/std/room";

void create() {
    ::create();
    set_property("light", 2);
    set_property("indoors", 1);
    set("short", "Descartes' Workroom");
    set("long", "This is where Descartes works.\nIt is a cube.\n");
    set_exits( ({ "/d/standard/square" }), ({ "square" }) );
}
-----
```

If you understand the entire textbook to this point, you should recognize of the code the following:
1) create() is the definition of a function (hey! he did not declare it)
2) It makes calls to set_property(), set(), and set_exits(), none of which are declared or defined in the code.
3) There is a line at the top that is no variable or function declaration nor is it a function definition!

This chapter will seek to answer the questions that should be in your head at this point:
1) Why is there no declaration of create()?
2) Where are the functions set_property(), set(), and set_exits() declared and defined?
3) What the hell is that line at the top of the file?

5.2 Object oriented programming
Inheritance is one of the properties which define true object oriented programming (OOP). It allows you to create generic code which can be used in many different ways by many different programs. What a mudlib does is create these generalized files (objects) which you use to make very specific objects.

If you had to write the code necessary for you to define the workroom above, you would have to write about 1000 lines of code to get all the functionality of the room above. Clearly that is a waste of disk space. In addition, such code does not interact well with players and other rooms since every creator is making up his or her own functions to perform the functionality of a room. Thus, what you might use to write out the room's long description,

query_long(), another wizard might be calling long(). This is the primary reason mudlibs are not compatible, since they use different protocols for object interaction.

OOP overcomes these problems. In the above workroom, you inherit the functions already defined in a file called "/std/room.c". It has all the functions which are commonly needed by all rooms defined in it. When you get to make a specific room, you are taking the general functionality of that room file and making a unique room by adding your own function, create().

5.3 How inheritance works

As you might have guessed by now, the line:

```
-----
inherit "/std/room";
-----
```

has you inherit the functionality of the room "/std/room.c". By inheriting the functionality, it means that you can use the functions which have been declared and defined in the file "/std/room.c" In the Nightmare Mudlib, "/std/room.c" has, among other functions, set_property(), set(), and set_exits() declared and defined. In your function create(), you are making calls to those functions in order to set values you want your room to start with. These values make your room different from others, yet able to interact well with other objects in memory.

In actual practice, each mudlib is different, and thus requires you to use a different set of standard functions, often to do the same thing. It is therefore beyond the scope of this textbook even to describe what functions exist and what they do. If your mudlib is well documented, however, then (probably in /doc/build) you will have tutorials on how to use the inheritable files to create such objects. These tutorials should tell you what functions exist, what input they take, the data type of their output, and what they do.

5.4 Chapter summary

This is far from a complete explanation of the complex subject of inheritance. The idea here is for you to be able to understand how to use inheritance in creating your objects. A full discussion will follow in a later textbook. Right now you should know the following:

1) Each mudlib has a library of generic objects with their own general functions used by creators through inheritance to make coding objects easier and to make interaction between objects smoother.
2) The functions in the inheritable files of a mudlib vary from mudlib to mudlib. There should exist documentation on your mud on how to use each inheritable file. If you are unaware what functions are available, then there is simply no way for you to use them. Always pay special attention to the data types of the input and the data types of ay output.
3) You inherit the functionality of another object through the line:

```
-----
inherit "filename";
-----
```

where filename is the name of the file of the object to be inherited.

```
       This line goes at the beginning of your code.
```

Note:
You may see the syntax ::create() or ::init() or ::reset() in places.
You do not need fully to understand at this point the full nuances of this,
but you should have a clue as to what it is. The "::" operator is a way
to call a function specifically in an inherited object (called the scope
resolution operator). For instance, most muds' room.c has a function
called create(). When you inherit room.c and configure it, you are doing
what is called overriding the create() function in room.c. This means
that whenever ANYTHING calls create(), it will call *your* version and not
the one in room.c. However, there may be important stuff in the room.c
version of create(). The :: operator allows you to call the create() in
room.c instead of your create().
An example:

```
-----
#1

inherit "/std/room";

void create() { create(); }
-----

-----
#2

inherit "/std/room";

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

Example 1 is a horror. When loaded, the driver calls create(), and then
create() calls create(), which calls create(), which calls create()...
In other words, all create() does is keep calling itself until the driver
detects a too deep recursion and exits.

Example 2 is basically just a waste of RAM, as it is no different from room.c
functionally. With it, the driver calls its create(), which in turn calls
::create(), the create() in room.c. Otherwise it is functionally
exactly the same as room.c.

CHAPTER 6: Variable Handling

6.1 Review
By now you should be able to code some simple objects using your muds standard
object library. Inheritance allows you to use functions defined in those
objects without having to go and define yourself. In addition,
you should know how to declare your own functions. This
chapter will teach you about the basic elements of LPC which will allow you to
define your own functions using the manipulation of variables.

6.2 Values and objects
Basically, what makes objects on the mud different are two things:
1) Some have different functions
2) All have different values

Now, all player objects have the same functions. They are therefore differentiated by the values they hold. For instance, the player named "Forlock" is different from "Descartes" *at least* in that they have different values for the variable true_name, those being "descartes" and "forlock".

Therefore, changes in the game involve changes in the values of the objects in the game. Functions are used to name specific process for manipulating values. For instance, the create() function is the function whose process is specifically to initialize the values of an object.
Within a function, it is specifically things called instructions which are responsible for the direct manipulation of variables.

6.3 Local and global variables
Like variables in most programming language, LPC variables may be declared as variables "local" to a specific function, or "globally" available to all functions. Local variables are declared inside the function which will use them. No other function knows about their existence, since the values are only stored in memory while that function is being executed.
A global variable is available to any function which comes after its declaration in the object code. Since global variables take up RAM for the entire existence of the object, you should use them only when you need a value stored for the entire existence of the object.
Have a look at the following 2 bits of code:

```
-----
int x;

int query_x() { return x; }

void set_x(int y) { x = y; }
-----
```

```
-----
void set_x(int y) {
    int x;

    x = y;
    write("x is set to x"+x+" and will now be forgotten.\n");
}
-----
```

In the first example, x is declared outside of any functions, and therefore will be available to any function declared after it. In that example, x is a global variable.
In the second example, x is declared inside the function set_x(). It only exists while the function set_x() is being executed. Afterwards, it ceases to exist. In that example, x is a local variable.

6.4 Manipulating the values of variables
Instructions to the driver are used to manipulate the values of variables. An example of an instruction would be:

```
-----
x = 5;
-----
```

The above instruction is self-explanatory. It assigns to the variable
x the value 5. However, there are some important concepts in involved
in that instruction which are involved in instructions in general.
The first involves the concept of an expression. An expression is
any series of symbols which have a value. In the above instruction,
the variable x is assigned the value of the expression 5. Constant
values are the simplest forms in which expressions can be put. A constant
is a value that never changes like the int 5 or the string "hello".
The last concept is the concept of an operator. In the above example,
the assignment operator = is used.

There are however many more operators in LPC, and expressions can get
quite complex. If we go up one level of complexity, we get:

```
-----
y = 5;
x = y +2;
-----
```

The first instruction uses the assignment operator to assign the value
of the constant expression 5 to the variable y. The second one
uses the assignment operator to assign to x the value of the expression
(y+2) which uses the addition operator to come up with a value which
is the sum of the value of y and the value of the constant expression 2.
Sound like a lot of hot air?

In another manner of speaking, operators can be used to form complex
expressions. In the above example, there are two expressions in the
one instruction x = y + 2;:
    1) the expression y+2
    2) the expression x = y + 2
As stated before, all expressions have a value. The expression
y+2 has the value of the sum of y and 2 (here, 7);
The expression x = y + 2 *also* has the value of 7.
So operators have to important tasks:
    1) They *may* act upon input like a function
    2) They evaluate as having a value themselves.
Now, not all operators do what 1 does. The = operators does act upon
the value of 7 on its right by assigning that value to x. The operator
+ however does nothing. They both, however, have their own values.

6.5 Complex expressions

As you may have noticed above, the expression x = 5 *itself* has a value
of 5. In fact, since LPC operators themselves have value as expressions,
they cal allow you to write some really convoluted looking nonsense like:

```
    i = ( (x=sizeof(tmp=users())) ? --x : sizeof(tmp=children("/std/monster"))-1)
```

which says basically:
    assing to tmp the array returned by the efun users(), then assign to x
    the value equal to the number of elements to that array. If the value
    of the expression assigning the value to x is true (not 0), then assign
    x by 1 and assign the value of x-1 to i. If x is false though,
    then set tmp to the array returned by the efun children(), and then
    assign to i the value of the number of members in the array tmp -1.
Would you ever use the above statement? I doubt it. However you might
see or use expressions similar to it, since the ability to consolidate
so much information into one single line helps to speed up the execution of

```
your code.  A more often used version of this property of LPC operators
would be something like:
    x = sizeof(tmp = users());
    while(i--) write((string)tmp[i]->query_name()+"\n");
instead of writing something like:
    tmp = users();
    x = sizeof(tmp);
    for(i=0; iquery_name()+"\n");
Things like for(), while(), arrays and such will be explained later.
But the first bit of code is more concise and it executed faster.

NOTE: A detailed description of all basic LPC operators follows the chapter
summary.
```

6.6 Chapter Summary

```
You now know how to declare variables and understand the difference between
declaring and using them globally or locally.  Once you become familiar
with your driver's efuns, you can display those values in many different
ways.  In addition, through the LPC operators, you know how to change
and evaluate the values contained in variables.  This is useful of course
in that it allows you to do something like count how many apples have
been picked from a tree, so that once all apples have been picked, no
players can pick more.  Unfortunately, you do not know how to have
code executed in anything other than a linera fashion.  In other words,
hold off on that apple until the next chapter, cause you do not know
how to check if the apples picked is equal to the number of apples in the
tree.  You also do not know about the special function init() where you
give new commands to players.  But you are almost ready to code a nice,
fairly complex area.
```

6.7 LPC operators

```
This section contains a detailed listing of the simpler LPC operators,
including what they do to the values they use (if anything) and the value
that they have.

The operators described here are:
=    +    -    *    /    %    +=    -=    *=    /=    %=
--    ++    ==    !=    >    <    >=    <=    !    &&    ||
->    ? :

Those operators are all described in a rather dry manner below, but it is best
to at least look at each one, since some may not behave *exactly* as
you think.  But it should make a rather good reference guide.

= assignment operator:
    example: x = 5;
    value: the value of the variable on the *left* after its function is done
    explanation: It takes the value of any expression on the *right* and
      assigns it to the variable on the *left*.  Note that you must use
      a single variable on the left, as you cannot assign values to
      constants or complex expressions.

+ addition operator:
    example: x + 7
    value: The sum of the value on the left and the value on the right
    exaplanation: It takes the value of the expression on the right and
      adds it to the value of the expression on the left. For values
```

```
      of type int, this means the numerical sum.  For strings,
      it means that the value on the right is stuck onto the value on
      the left ("ab" is the value of "a"+"b").  This operator does not
      modify any of the original values (i.e. the variable x from
      above retains its old value).

- subtraction operator:
    example: x - 7
    value: the value of the expression on the left reduced by the right
    explanation:  Same characteristics as addition, except it subtracts.
      With strings: "a" is the value of "ab" - "b"

* multiplication operator:
    example: x*7
    value and explanation: same as with adding and subtracting except
      this one performs the math of multiplication

/ division operator:
    example: x/7
    value and explanation: see above

+= additive assignment operator:
    example: x += 5
    value: the same as x + 5
    exaplanation: It takes the value of the variable on the left
      and the value of the expression on the right, adds them together
      and assigns the sum to the variable on the left.
      example: if x = 2... x += 5 assigns the value
        7 to the variable x.  The whole expression
        has the value of 7.

-= subtraction assignment operator
    example: x-=7
    value: the value of the left value reduced by the right value
    examplanation: The same as += except for subtraction.

*= multiplicative assignment operator
    example: x *= 7
    value: the value of the left value multiplied by the right
    explanation: Similar to -= and += except for addition.

/= division assignment operator
    example: x /= 7
    value: the value of the variable on the left divided by the right value
    explanation: similar to above, except with division

++ post/pre-increment operators
    examples: i++ or ++i
    values:
      i++ has the value of i
      ++i has the value of i+1
    explanation: ++ changes the value of i by increasing it by 1.
      However, the value of the expression depends on where you
      place the ++.  ++i is the pre-increment operator.  This means
      that it performs the increment *before* giving a value.
      i++ is the post-ncrement operator.  It evalutes before incrementing
      i.  What is the point?  Well, it does not much matter to you at
```

```
      this point, but you should recognize what it means.

-- post/pre-decrement operators
    examples: i-- or --i
    values:
      i-- the value of i
      --i the value of i reduced by 1
    explanation: like ++ except for subtraction

== equality operator
    example: x == 5
    value: true or false (not 0 or 0)
    explanation: it does nothing to either value, but
      it returns true if the 2 values are the same.
      It returns false if they are not equal.

!= inequality operator
    example: x != 5
    value: true or false
    explanation returns true if the left expression is not equal to the right
      expression.  It returns fals if they are equal

> greater than operator
    example: x > 5
    value: true or false
    explanation: true only if x has a value greater than 5
      false if the value is equal or less

< less than operator
>= greater than or equal to operator
<= less than or equal to operator
    examples: x < y    x >= y    x <= y
    values: true or false
    explanation: similar as to > except
      < true if left is less than right
      >= true if left is greater than *or equal to* right
      <= true if the left is less than *or equal to* the right

&& logical and operator
|| logical or operator
    examples: x && y      x || y
    values: true or false
    explanation: If the right value and left value are non-zero, && is true.
      If either are false, then && is false.
      For ||, only one of the values must be true for it to evaluate
      as true.  It is only false if both values indeed
      are false

! negation operator
    example: !x
    value: true or false
    explanation: If x is true, then !x is false
      If x is false, !x is true.

A pair of more complicated ones that are here just for the sake of being
here.  Do not worry if they utterly confuse you.
```

```
-> the call other operator
    example: this_player()->query_name()
    value: The value returned by the function being called
    explanation:  It calls the function which is on the right in the object
      on the left side of the operator.  The left expression *must* be
      an object, and the right expression *must* be the name of a function.
      If not such function exists in the object, it will return 0 (or
      more correctly, undefined).

? : conditional operator
    example: x ? y : z
    values: in the above example, if x is try, the value is y
      if x is false, the value of the expression is z
    explanation: If the leftmost value is true, it will give the expression as
      a whole the value of the middle expression.  Else, it will give the
      expression as a whole the value of the rightmost expression.

A note on equality:  A very nasty error people make that is VERY difficult
to debug is the error of placing = where you mean ==.  Since
operators return values, they both make sense when being evaluated.
In other words, no error occurs.  But they have very different values.  For example:
  if(x == 5)    if(x = 5)
The value of x == 5 is true if the value of x is 5, false othewise.
The value of x = 5 is 5 (and therefore always true).
The if statement is looking for the expression in () to be either true or false,
so if you had = and meant ==, you would end up with an expression that is
always true.  And you would pull your hair out trying to figure out
why things were not happening like they should :)
```

CHAPTER 7: Flow Control

7.1 Review of variables

Variables may be manipulated by assigning or changing values with the expressions =, +=, -=, ++, --. Those expressions may be combined with the expressions -, +, *, /, %. However, so far, you have only been shown how to use a function to do these in a linear way. For example:

```
int hello(int x) {
    x--;
    write("Hello, x is "+x+".\n");
    return x;
}
```

is a function you should know how to write and understand. But what if you wanted to write the value of x only if x = 1? Or what if you wanted it to keep writing x over and over until x = 1 before returning? LPC uses flow control in exactly the same way as C and C++.

7.2 The LPC flow control statements

LPC uses the following expressions:

```
if(expression) instruction;

if(expression) instruction;
else instruction;
```

```
if(expression) instruction;
else if(expression) instruction;
else instruction;

while(expression) instruction;

do { instruction; } while(expression);

switch(expression) {
    case (expression): instruction; break;
    default: instruction;
}
```

Before we discuss these, first something on what is meant by expression and instruction. An expression is anything with a value like a variable, a comparison (like x>5, where if x is 6 or more, the value is 1, else the value is 0), or an assignment(like x += 2). An instruction can be any single line of lpc code like a function call, a value assignment or modification, etc.

You should know also the operators &&, ||, ==, !=, and !. These are the logical operators. They return a nonzero value when true, and 0 when false. Make note of the values of the following expressions:

```
(1 && 1) value: 1   (1 and 1)
(1 && 0) value: 0   (1 and 0)
(1 || 0) value: 1   (1 or 0)
(1 == 1) value: 1   (1 is equal to 1)
(1 != 1) value: 0   (1 is not equal to 1)
(!1) value: 0       (not 1)
(!0) value: 1       (not 0)
```

In expressions using &&, if the value of the first item being compared is 0, the second is never tested even. When using ||, if the first is true (1), then the second is not tested.

7.3 if()

The first expression to look at that alters flow control is if(). Take a look at the following example:

```
1 void reset() {
2     int x;
3
4     ::reset();
5     x = random(10);
6     if(x > 50) set_search_func("floorboards", "search_floor");
7 }
```

The line numbers are for reference only.
In line 2, of course we declare a variable of type int called x. Line 3 is aethetic whitespace to clearly show where the declarations end and the function code begins. The variable x is only available to the function reset().
Line 4 makes a call to the room.c version of reset().
Line 5 uses the driver efun random() to return a random number between 0 and the parameter minus 1. So here we are looking for a number between 0 and 99.

In line 6, we test the value of the expression (x>50) to see if it is true or false. If it is true, then it makes a call to the room.c function set_search_func(). If it is false, the call to set_search_func() is never executed.
In line 7, the function returns driver control to the calling function (the driver itself in this case) without returning any value.

If you had wanted to execute multiple instructions instead of just the one, you would have done it in the following manner:

```
if(x>50) {
    set_search_func("floorboards", "search_floor");
    if(!present("beggar", this_object())) make_beggar();
}
```

Notice the {} encapsulate the instructions to be executed if the test expression is true. In the example, again we call the room.c function which sets a function (search_floor()) that you will later define yourself to be called when the player types "search floorboards" (NOTE: This is highly mudlib dependent. Nightmare mudlibs have this function call. Others may have something similar, while others may not have this feature under any name). Next, there is another if() expression that tests the truth of the expression (!present("beggar",this_object())). The ! in the test expression changes the truth of the expression which follows it. In this case, it changes the truth of the efun present(), which will return the object that is a beggar if it is in the room (this_object()), or it will return 0 if there is no beggar in the room. So if there is a beggar still living in the room, (present("beggar", this_object())) will have a value equal to the beggar object (data type object), otherwise it will be 0. The ! will change a 0 to a 1, or any nonzero value (like the beggar object) to a 0. Therefore, the expression (!present("beggar", this_object())) is true if there is no beggar in the room, and false if there is. So, if there is no beggar in the room, then it calls the function you define in your room code that makes a new beggar and puts it in the room. (If there is a beggar in the room, we do not want to add yet another one :))

Of course, if()'s often comes with ands or buts :). In LPC, the formal reading of the if() statement is:

```
if(expression) { set of intructions }
else if(expression) { set of instructions }
else { set of instructions }
```

This means:

If expression is true, then do these instructions.
Otherise, if this second expression is true, do this second set.
And if none of those were true, then do this last set.

You can have if() alone:

```
if(x>5) write("Foo,\n");
```

with an else if():

```
if(x > 5) write("X is greater than 5.\n");
```

```
else if(x >2) write("X is less than 6, but greater than 2.\n");
```

with an else:

```
if(x>5) write("X is greater than 5.\n");
else write("X is less than 6.\n");
```

or the whole lot of them as listed above. You can have any number of else if()'s in the expression, but you must have one and only one if() and at most one else. Of course, as with the beggar example, you may nest if() statements inside if() instructions. (For example,

```
    if(x>5) {
        if(x==7) write("Lucky number!\n");
        else write("Roll again.\n");
    }
    else write("You lose.\n");
```

7.4 The statements: while() and do {} while()

Prototype:

```
while(expression) { set of instructions }
do { set of instructions } while(expression);
```

These allow you to create a set of instructions which continue to execute so long as some expression is true. Suppose you wanted to set a variable equal to a player's level and keep subtracting random amounts of either money or hp from a player until that variable equals 0 (so that player's of higher levels would lose more). You might do it this way:

```
1    int x;
2
3    x = (int)this_player()->query_level();  /* this has yet to be explained */
4    while(x > 0) {
5        if(random(2)) this_player()->add_money("silver", -random(50));
6        else this_player()->add_hp(-(random(10));
7        x--;
8    }
```

The expression this_player()->query_level() calIn line 4, we start a loop that executes so long as x is greater than 0.

Another way we could have done this line would be:

```
        while(x) {
```

The problem with that would be if we later made a change to the funtion
y anywhere between 0 and 49 coins.
In line 6, if instead it returns 0, we call the add_hp() function in the player which reduces the player's hit points anywhere between 0 and 9 hp.
In line 7, we reduce x by 1.
At line 8, the execution comes to the end of the while() instructions and goes back up to line 4 to see if x is still greater than 0. This loop will keep executing until x is finally less than 1.

You might, however, want to test an expression *after* you execute some instructions. For instance, in the above, if you wanted to execute the instructions at least once for everyone, even if their level is below the test level:

```
    int x;
```

```
    x = (int)this_player()->query_level();
    do {
        if(random(2)) this_player()->add_money("silver", -random(50));
        else this_player()->add_hp(-random(10));
        x--;
    } while(x > 0);
```

This is a rather bizarre example, being as few muds have level 0 players.
And even still, you could have done it using the original loop with
a different test. Nevertheless, it is intended to show how a do{} while()
works. As you see, instead of initiating the test at the beginning of the
loop (which would immediately exclude some values of x), it tests after
the loop has been executed. This assures that the instructions of the loop
get executed at least one time, no matter what x is.

7.5 for() loops
Prototype:
for(initialize values ; test expression ; instruction) { instructions }

initialize values:
This allows you to set starting values of variables which will be used
in the loop. This part is optional.

test expression:
Same as the expression in if() and while(). The loop is executed
as long as this expression (or expressions) is true. You must have a
test expression.

instruction:
An expression (or expressions) which is to be executed at the end of each
loop. This is optional.

Note:
for(;expression;) {}
IS EXACTLY THE SAME AS
while(expression) {}

Example:

```
1    int x;
2
3    for(x= (int)this_player()->query_level(); x>0; x--) {
4        if(random(2)) this_player()->add_money("silver", -random(50));
5        else this_player()->add_hp(-random(10));
6    }
```

This for() loop behaves EXACTLY like the while() example.
Additionally, if you wanted to initialize 2 variables:

for(x=0, y=random(20); x7.6 The statement: switch()
Prototype:

```
switch(expression) {
    case constant: instructions
    case constant: instructions
    ...
    case constant: instructions
```

```
    default: instructions
}
```

This is functionally much like if() expressions, and much nicer to the
CPU, however most rarely used because it looks so damn complicated.
But it is not.

First off, the expression is not a test. The cases are tests. A English
sounding way to read:

```
1    int x;
2
3    x = random(5);
4    switch(x) {
5        case 1: write("X is 1.\n");
6        case 2: x++;
7        default: x--;
8    }
9    write(x+"\n");
```

is:

set variable x to a random number between 0 and 4.
In case 1 of variable x write its value add 1 to it and subtract 1.
In case 2 of variable x, add 1 to its value and then subtract 1.
In other cases subtract 1.
Write the value of x.

switch(x) basically tells the driver that the variable x is the value
we are trying to match to a case.
Once the driver finds a case which matches, that case *and all following
cases* will be acted upon. You may break out of the switch statement
as well as any other flow control statement with a break instruction in
order only to execute a single case. But that will be explained later.
The default statement is one that will be executed for any value of
x so long as the switch() flow has not been broken. You may use any
data type in a switch statement:

```
string name;

name = (string)this_player()->query_name();
switch(name) {
    case "descartes": write("You borg.\n");
    case "flamme":
    case "forlock":
    case "shadowwolf": write("You are a Nightmare head arch.\n");
    default: write("You exist.\n");
}
```

For me, I would see:
You borg.
You are a Nightmare head arch.
You exist.

Flamme, Forlock, or Shadowwolf would see:
You are a Nightmare head arch.
You exist.

```
Everyone else would see:
You exist.
```

7.7 Altering the flow of functions and flow control statements

```
The following instructions:
return    continue    break

alter the natural flow of things as described above.
First of all,
return
no matter where it occurs in a function, will cease the execution of that
function and return control to the function which called the one the
return statement is in.  If the function is NOT of type void, then a
value must follow the return statement, and that value must be of a
type matching the function.  An absolute value function would look
like this:

int absolute_value(int x) {
    if(x>-1) return x;
    else return -x;
}

In the second line, the function ceases execution and returns to the calling
function because the desired value has been found if x is a positive
number.

continue is most often used in for() and while statements.  It serves
to stop the execution of the current loop and send the execution back
to the beginning of the loop.  For instance, say you wanted to avoid
division by 0:

x= 4;
while( x > -5) {
    x--
    if(!x) continue;
    write((100/x)+"\n");
}
write("Done.\n")

You would see the following output:
33
50
100
-100
-50
-33
-25
Done.
To avoid an error, it checks in each loop to make sure x is not 0.
If x is zero, then it starts back with the test expression without
finishing its current loop.

In a for() expression
 for(x=3; x>-5; x--) {
    if(!x) continue;
    write((100/x)+"\n");
```

```
 }
 write("Done.\n");
```
It works much the same way. Note this gives exactly the same output
as before. At x=1, it tests to see if x is zero, it is not, so it
writes 100/x, then goes back to the top, subtracts one from x, checks to
see if it is zero again, and it is zero, so it goes back to the top
and subtracts 1 again.

break
This one ceases the function of a flow control statement. No matter
where you are in the statement, the control of the program will go
to the end of the loop. So, if in the above examples, we had
used break instead of continue, the output would have looked like this:

```
33
50
100
Done.
```

continue is most often used with the for() and while() statements.
break however is mostly used with switch()

```
switch(name) {
    case "descartes": write("You are borg.\n"); break;
    case "flamme": write("You are flamme.\n"); break;
    case "forlock": write("You are forlock.\n"); break;
    case "shadowwolf": write("You are shadowwolf.\n"); break;
    default: write("You will be assimilated.\n");
}
```

This functions just like:

```
if(name == "descartes") write("You are borg.\n");
else if(name == "flamme") write("You are flamme.\n");
else if(name == "forlock") write("You are forlock.\n");
else if(name == "shadowwolf") write("You are shadowwolf.\n");
else write("You will be assimilated.\n");
```

except the switch statement is much better on the CPU.
If any of these are placed in nested statements, then they alter the
flow of the most immediate statement.

7.8 Chapter summary
This chapter covered one hell of a lot, but it was stuff that needed to
be seen all at once. You should now completely understand if() for()
while() do{} while() and switch(), as well as how to alter their flow
using return, continue, and break. Effeciency says if it can be done in
a natural way using switch() instead of a lot of if() else if()'s, then
by all means do it. You were also introduced to the idea of calling
functions in other objects. That however, is a topic to be detailed later.
You now should be completely at ease writing simple rooms (if you have
read your mudlib's room building document), simple monsters, and
other sorts of simple objects.

CHAPTER 8: The data type "object"

8.1 Review
You should now be able to do anything so long as you stick to calling functions within your own object. You should also know, that at the bare minimum you can get the create() (or reset()) function in your object called to start just by loading it into memory, and that your reset() function will be called every now and then so that you may write the code necessary to refresh your room. Note that neither of these functions MUST be in your object. The driver checks to see if the function exists in your object first. If it does not, then it does not bother. You are also acquainted with the data types void, int, and string.

8.2 Objects as data types
In this chapter you will be acquainted with a more complex data type, object. An object variable points to a real object loaded into the driver's memory. You declare it in the same manner as other data types:

```
    object ob;
```

It differs in that you cannot use +, -, +=, -=, *, or / (what would it mean to divide a monster by another monster?). And since efuns like say() and write() only want strings or ints, you cannot write() or say() them (again, what would it mean to say a monster?).
But you can use them with some other of the most important efuns on any LPMud.

8.3 The efun: this_object()
This is an efun which returns an object in which the function being executed exists. In other words, in a file, this_object() refers to the object your file is in whether the file gets cloned itself or inherted by another file. It is often useful when you are writing a file which is getting inherited by another file. Say you are writing your own living.c which gets inherited by user.c and monster.c, but never used alone. You want to log the function set_level() it is a player's level being set (but you do not care if it is a monster.
You might do this:

```
void set_level(int x) {
    if(this_object()->is_player()) log_file("levels", "foo\n");
    level = x;
}
```

Since is_player() is not defined in living.c or anything it inherits, just saying if(is_player()) will result in an error since the driver does not find that function in your file or anything it inherits. this_object() allows you to access functions which may or may not be present in any final products because your file is inherited by others without resulting in an error.

8.4 Calling functions in other objects
This of course introduces us to the most important characteristic of the object data type. It allows us to access functions in other objects. In previous examples you have been able to find out about a player's level, reduce the money they have, and how much hp they have.
Calls to functions in other objects may be done in two ways:

```
object->function(parameters)
call_other(object, "function", parameters);
```

example:

```
this_player()->add_money("silver", -5);
call_other(this_player(), "add_money", "silver", -5);
```

In some (very loose sense), the game is just a chain reaction of function calls initiated by player commands. When a player initiates a chain of function calls, that player is the object which is returned by the efun this_player(). So, since this_player() can change depending on who initiated the sequence of events, you want to be very careful as to where you place calls to functions in this_player(). The most common place you do this is through the last important lfun (we have mentioned create() and reset()) init().

8.5 The lfun: init()

Any time a living thing encounters an object (enters a new room, or enters the same room as a certain other object), init() is called in all of the objects the living being newly encounters. It is at this point that you can add commands the player can issue in order to act. Here is a sample init() function in a flower.

```
void init() {
    ::init();
    add_action("smell_flower", "smell");
}
```

Ito smell_flower(). So you should have smell_flower() look like this:

```
1 int smell_flower(string str);        /* action functions are type int */
2
3 int smell_flower(string str) {
4    if(str != "flower") return 0;     /* it is not the flower being smelled */
5    write("You sniff the flower.\n");
6    say((string)this_player()->query_cap_name()+" smells the flower.\n");
7    this_player()->add_hp(random(5));
8    return 1;
9 }
```

In line 1, we have our function declared.
In line 3, smell_flower() begins. str becomes whatever comes after the players command (not including the first white space).
In line 4, it checks to see if the player had typed "smell flower". If the player had typed "smell cheese", then str would be "cheese". If it is not in fact "flower" which is being smelled, then 0 is returned, letting the driver know that this was not the function which should have been called. If in fact the player had a piece of cheese as well which had a smell command to it, the driver would then call the function for smelling in that object. The driver will keep calling all functions tied to smell commands until one of them returns 1. If they all return 0, then the player sees "What?"
In line 5, the efun write() is called. write() prints the string which is passed to it to this_player(). So whoever typed the command here sees "You sniff the flower."
In line 6, the efun say() is called. say() prints the string which is doing the sniffing, we have to call the query_cap_name() function in this_player(). That way if the player is invis, it will say "Someone" (or something like that), and it will also be properly capitalized.
In line 7, we call the add_hp() function in the this_player() object,

since we want to do a little healing for the sniff (Note: do not
code this object on your mud, whoever balances your mud will shoot you).
In line 8, we return control of the game to the driver, returning 1 to
let it know that this was in fact the right function to call.

8.6 Adding objects to your rooms

And now, using the data type object, you can add monsters to your rooms:

```
void create() {
    ::create();
    set_property("light", 3);
    set("short", "Krasna Square");
    set("long", "Welcome to the Central Square of the town of Praxis.\n");
    set_exits( ({ "d/standard/hall" }), ({ "east" }) );
}

void reset() {
    object ob;

    ::reset();
    if(present("guard")) return;     /* Do not want to add a guard if */
    ob = new("/std/monster");        /* one is already here           */
    ob->set_name("guard");
    ob->set("id", ({ "guard", "town guard" }) );
    ob->set("short", "Town guard");
    ob->set("long", "He guards Praxis from nothingness.\n");
    ob->set_gender("male");
    ob->set_race("human");
    ob->set_level(10);
    ob->set_alignment(200);
    ob->set_humanoid();
    ob->set_hp(150);
    ob->set_wielding_limbs( ({ "right hand", "left hand" }) );
    ob->move(this_object());
}
```

Now, this will be wildly different on most muds. Some, as noted before,
in that object so you have a uniquely configured monster object. The
last act in native muds is to call move() in the monster object to move
it to this room (this_object()). In compat muds, you call the efun
move_object() which takes two parameters, the object to be moved, and the
object into which it is being moved.

8.7 Chapter summary

At this point, you now have enough knowledge to code some really nice
stuff. Of course, as I have been stressing all along, you really need
to read the documents on building for your mud, as they detail which
functions exist in which types of objects for you to call. No matter
what your knowledge of the mudlib is, you have enough know-how to
give a player extra things to do like sniffing flowers or glue or whatever.
At this point you should get busy coding stuff. But the moment things
even look to become tedious, that means it is time for you to move to
the next level and do more. Right now code yourself a small area.
Make extensive use of the special functions coded in your mud's
room.c (search the docs for obscure ones no one else seems to use).
Add lots o' neat actions. Create weapons which have magic powers which
gradually fade away. All of this you should be able to do now. Once

this becomes routine for you, it will be time to move on to intermediate stuff. Note that few people actually get to the intermediate stuff. If you have played at all, you notice there are few areas on the mud which do what I just told you you should be able to do. It is not because it is hard, but because there is a lot of arrogance out there on the part of people who have gotten beyond this point, and very little communicating of that knowledge. The trick is to push yourself and think of something you want to do that is impossible. If you ask someone in the know how to do X, and they say that is impossible, find out youself how to code it by experimenting.

George Reese
Descartes of Borg
12 july 1993
borg@hebron.connected.com
Descartes@Nightmare (intermud)
Descartes@Igor (not intermud)

Table of contents