LPC

Table of Contents

Table of Contents

Table of Contents

LPC

Copying Conditions

This tutorial was produced in good faith for use by people who like to program muds for their own pleasure. That means that I won't charge for using or distributing it provided that this *is* the purpose it's being used for.

Specifically, I want to make sure that those who run commercial muds don't provide this manual as an aid to further their monetary aims. If they want it they can either pay me a lot of money for it or produce one of their own.

Please read the copyright statement in the printed section of this manual to get the full text. The gist of it, however, is exactly what I just related here.

Introduction

This tutorial is intended to be something that anyone can read and learn from. Now, this is, of course, impossible, so let's amend that a bit. It's a tutorial that anyone with at least a bit of knowledge of programming and a will to learn can use. You don't have to know about C before you start, and I believe that even true virgins might be able to learn how to code. They will, of course, have a much harder job though.

Experienced coders, even mud coders, will need to read the tutorial since it explains concepts that are unique for this game, but they will be able to skim most of it and focus only on the trickier bits. I leave the selection of what is and what is not important to you, the reader, since you're the only one who knows how much you need to learn.

As the LPC language in actual application is closely knitted to the mudlib which it is used in, I will also cover part of that. However, I will *not* teach you how to use the mudlib in detail, you'll have to read other documents for that. All of this makes this tutorial pretty specific for games, and Genesis in particular. Keep this in mind if you are reading it on another game.

I hope you'll find the tutorial useful and worth the effort it takes to read. It sure took a lot of effort to write, so it'd better not be for nothing! :)

---

i − Acknowledgments

I'd like to start by thanking Thorsten Lockert and Christian Markus, perhaps better known as Plugh and Olorin, for their help in proofreading this text, suggesting improvements and in general being supportive and providing constructive feedback.

Without them this rather lengthy project would have taken even longer (mind−boggling, but true) to complete and ended up a *lot* worse.

---

ii − Tutorial Setup

The manual is divided into three progressively more advanced chapters. The first chapter will be explain basics about coding and LPC without delving too deep into specifics.

The second chapter will be for a more intermediate level of coding, explaining in full all the aspects of functions and operators that might have been treated a bit too easy in the first chapter.

The third and final chapter handles whatever might be left when chapter two is done. Well, not *everything*; the tutorial will not explain all the intricacies of the gamedriver and the mudlib. If you are looking for info about creating your own mudlib or doing very advanced stuff you'll have to learn that from reading the actual code.

If you are a newbie wizard you might feel a bit taken back at first, looking at this rather thick tutorial. However, it's quite necessary that you read *all* of it and leave nothing for the future. You will undoubtably have to at least recognize subjects from all three chapters, even though mostly you will actually only use the information in the first two. Heck, there's a lot of old wizards out there who hardly even master the first one, a scary thought! Among other things that's one of the fundamental reasons why I'm writing this manual...

The manual is fairly extensive, but it's for learning LPC for domain coding *only*. This means that it's not a complete LPC reference manual. Some of the more esoteric and unusual efuns and functionalities are not covered since they only would be used by mudlib coders or gamedriver hackers. This manual is not intended for them. Sorry, if that's what you were looking for you'll have to keep on searching for another source.

A small note about gender. Throughout the text I've used male pronouns. This is not because I scoff the thought of female coders, it's because the English language is male−oriented. Fact of life, like it or not. I guess I *could* have added a `(or she)' comment after all occurrences of the male `he', but that strikes me as more than just a bit silly. Until the English language comes up with a strictly neutral pronoun I'll stick to using the all−inclusive male one.

Chapters that describe efuns/sfuns/lfuns/macros in detail have a subcaption with the names of the discussed items within brackets for easy finding if you're just browsing the tutorial later.

---

iii − History of LPC

LPC is the interpreted mud language created by Lars Pensjoe for his invention LPMUD, an interactive multiuser environment suited for many purposes, games not the least among them. Since the first appearance in 1988 the language has changed dramatically.

Development was taken over by other people at Chalmers Datorfoerening, majorly Jakob Hallen, Lennart Augustsson and Anders Chrigstroem around 1990. They extended and refined the language extensively, but, as the name LPC hints of, it still retains its links to the language 'C'. The main differences lie in the object structure that's imposed on the language as well as some new data types to faciliate game programming. The language is not as free−form as 'C', but on the other hand it's more suitable for the purpose which it was created for − programming in a game environment.

---

iv − Gamedriver/Mudlib

The distinction between gamedriver and mudlib might seem hard to define at times, but it's really very simple. The gamedriver is the program that runs on the host computer. It is basically an interpreter in conjunction with an object management kernel, almost a small operating system in its own right. It defines and understands the LPC language, interprets it and executes the instructions given to it through the LPC objects in the game.

The mudlib is the collection of LPC objects that make up the basic game environment. While the gamedriver knows nothing about what it actually is doing, the mudlib does. (The mudlib can conversly be said to have no idea about how it does what it does, while the gamedriver does). It contains the basic set of LPC objects that are used in the game, the players, the monsters, the rooms etc. Individual efforts (domain code) are stored and handled separately and uses both the services of the gamedriver and the mudlib.

---

v − Administrative Setup

The game can be said to have been divided into three distinctive parts as suggested above; The gamedriver, the mudlib and the ***domain code***. The gamedriver and mudlib I have already explained. The domain concept is a way to organize the way code is written. A domain is principally a group of wizards working towards a predefined goal. This project can be limited in space, an actual area in the game world, or as intangiable as a guild or sect that the players can become members of.

Within a domain there is a leader, a domain Lord. He is the local taskmaster who decides what goes on and in which direction work should progress. In the domain all code is shared easily and usually is strongly interconnected.

Naturally there have to be ties between different domains as well, but these are usually weaker and actual code is seldom shared.

As a newbie wizard you will try to find a domain in which to enroll, that sounds interesting and inspiring to work with.

---

vi − Writing code

It might seem premature to tell you what your code should look like before you have learnt how to write it. However, it is a fundamental subject of great importance. Someone jokingly said that writing code correctly will make your teeth whiter, your hair darker and improve your sexlife. Well... it might not do that, but it'll certainly improve the overall quality of what you produce, at a very low cost. Mainly it's a matter of self−discipline.

Here are some good arguments for making the effort:

- It makes the code much more readable, not only for others, but also for yourself, particularly if you have to read or alter the code six months after you produced it.
- Since it is easier for others to read, and therefore easier to understand what you have done, it will be easier to help you in case of problems.
- Writing code properly actually makes it *better*, believe it or not. The reason for this is simply that writing code badly makes it easy to miss simple errors that get hidden among the crummy code.
- It can be real hard to find people willing to help you debug badly formatted code. I personally will ***not*** help people to debug code that looks too awful. The reason is that it's simply not worth the effort. With bad looking code there'll be lots of stupid errors (mostly misplaced or missing brackets) that'll turn up as soon as you start to indent the code properly.

What follows here is a rather lengthy instruction of how to put down your code in writing. Read it now even though you might not fully understand what is being discussed, then go back and re−read it later after having learnt the skills necessary. Doing that will make sure you'll remember the correct way of formatting your code.

1. One indent level is 4 spaces long, no more, no less. A new indent level is started at the beginning of each block.
2. Reserved words have a space after them, before the opening `(`, if any.

```
while (test)
    statement;
```

3. Block brackets start and end in the same column; the column of the first letter in a statement opening a block.

```
if (this)
{
    statement;
}
else if (that)
{
    another_statement;
}
else
{
    default_statement;
}
```

   Now, this is almost a religious matter for some coders. Representatives of another sect preaches that the block opening brace should be on the end of the line of the block statement and how you do this really isn't that important. Not as long as you do it the way I say, or you'll burn in COBOL hell forever :) No, seriously, pick one of the two ways of doing it and stick to it. The only *real* important thing is that you keep the indention−level straight throughout the code. If we all agree on something, it's that wavy indention−levels is something not to be tolerated.
4. Several arguments on a line separated by a comma have a space following the comma. `;`−separated lists and binary operators have a space both in front of, and after the operator.

```
int a, b, c;

for (a = 0 ; a < 10 ; a++)
{
    b = function_call(a, b * 2);
    c = b * 3 / 4;
}
```

5. If a loop statement has no body, put the ending `;` on a separate line.

```
while (!(var = func(var)))
    ;
```

   The reason for this is that if you should put it on the same line, it's *very* easy to miss real mistakes like this one just out of pure laziness:

```
for (i = 0 ; i < 100 ; i++);
{
    <code that gets executed only once, but always>
}
```

6. All `#define` and `#include` statements should be placed at the top of the file. It's possible to spread them out, but that will just be confusing.
7. The same goes for prototypes and global/static variables used in the file. Clump them all together, with a proper comment header, in the top of the file. It's possible to spread them out, but oh how easy it is to miss them when reading the code later...
8. Declarations of functions have the return type on a separate line from the function name.

```
public void
my_function(int a, int b)
{
```

```
    < code >
}
```

9. Break long lines of code in proper places so that they don't wrap on their own beyond the end of a 80−width screen. It looks ugly and becomes hard to read, not to mention print.
10. The file should begin with a proper header following this outline:

```
/*
 * <filename>
 *
 * <Short description of what the file does, no more than 5−7 lines.
 * ...
 * ... >
 *
 * Copyright (C): <your name and year>
 *
 */
```

    Read the game Copyright statement ***NOW*** in order to know what rules apply to code you produce for the game, in the game. It ought to reside in the file `/doc/COPYRIGHT'. If not, simply ask one of the game administrators.
11. Start ***every*** function with a header looking like this:

```
/*
 * Function name: <Function name>
 * Description:   <Short description of what the function does,
 *                 usually no more than three lines.
 * Arguments:     <A list of all arguments, one per line
 *                   arg1 − description no longer than the line.
 *                   arg2 − next argument, etc. >
 * Returns:       <What the function returns>
 */
```

    If the function doesn't take any arguments, or doesn't return anything, simply remove those lines in the header.
12. Put suitable comments in the code here and there when doing something that might look a bit obscure. Remember also that on your (assumed) level of competence, a lot of things are obscure :) Use your own judgement.

```
/*
 * Comment for code following this comment,
 * telling what it does
 */
< code >
```

13. Make sure all function names and local variables are written in lowercase alphabetical characters, possibly spacing words with an underscore (e.g. `function_name()`). Global variables should be written using the first letter of the word capitalized (e.g. `int GlobalTime;`). `#define`s should be written in capitals (e.g. `#define AMOEBA "one celled creature"`). Doing this makes it easy to see what kind of symbol is being handled at all times.

Now, the easiest way of getting the basic stuff done properly is to use the emacs editor, set up to use a modified c++ mode. The c++ mode understands about `::`−operators but needs a few hints on tabstops etc. Put these lines of code into your .emacs file and all will work just as it should:

```
;; emacs lisp script start
```

```
(setq auto-mode-alist (append '(
  ("\\.l" . my-c++-mode)
  ("\\.y" . my-c++-mode)
  ("\\.c" . my-c++-mode)
  ("\\.h" . my-c++-mode))
    auto-mode-alist))

(defun my-c++-mode () (interactive)
  (c++-mode)
  (setq c-indent-level 4)
  (setq c-brace-offset 0)
  (setq c-label-offset -4)
  (setq c-continued-brace-offset -4)
  (setq c-continued-statement-offset 4))

  ;; emacs end
```

An added advantage of using emacs is that later, when debugging other coder's attempts at writing code, correcting their horrible indention is as easy as typing 'M-<', 'M->', 'M-x indent-region'.

LPC basics

This chapter will teach you the very basics of programming, essential for understanding what follows. It will also introduce you to the concept of object oriented programming and explain some of the mudlib.

- Basic programming concepts: The most fundamental concepts of programming
- Basic LPC: LPC from the beginning
- The preprocessor: What it is and how it works

Basic programming concepts

We begin with he basic programming principles and the structure of LPC and the LPC environment.

- What is programming: A foundation in programming principles
- Compiled/Interpreted code: Explains the concepts
- Programs: How programs are set up
- Objects: The 'object' concept
- Object makeup: How objects are put together

What is programming?

This is a very philosophical question really. However, lets stick to the practical side and leave the zen bit to those who go for that kind of stuff.

Programming basically is the art of identifying a problem and putting the solution into symbols a computer can understand. A good programmer has a highly developed ability to see how a given problem can be split into smaller problems for which he has several solutions, and he also knows which particular solution he should pick to make the result as effective in terms of memory and speed as possible.

A programmer, as the previous passage suggests, actually tells the computer how to solve a problem. A computer can not come up with a solution to a problem itself. However, it's a lot quicker than you are, so problems that you can solve but would take you several lifetimes or simply just 'too long' are handled quickly by the computer.

What you need to learn is that special way of thinking that allows you to do this 'subdividing' thing where you see the steps needed to get from the beginning state to the solved state. You also need to learn the methods that make up these steps. Naturally this tutorial won't teach you 'how to program', it will only teach you the language used to put down the program in.

Who'll teach you programming then, in case you don't already know how to? Well... primarily other wizards in the game, and then yourself. Hard work in other words, there's *never* any shortcuts unfortunately, no matter what you need to learn. However, since this is a very amusing game let's hope you'll have great fun while acquiring the skills.

Compiled/Interpreted code

Programs are nothing but files containing instructions suited for the computer to understand. To program is to write lists of instructions in such a way that the computer reaches a predefined goal. Usually a program is compiled − translated − into a low−level code expressed in binary symbols (high and low electrical states in

computer memory) which the computer understands with ease. The actual language you use to program in is merely a convenient go−between, a compromise which both you and the computer can understand. The reason you compile code is that the translation step is fairly complicated and time−consuming. You'd rather just do that once and then store the result, using it directly over and over again.

LPC however, isn't compiled, it's interpreted. The instructions are read and translated to the computer one by one, executed and forgotten. Well, this is not 100% true. In fact, the gamedriver which is the program running on the host computer translates the LPC code into an intermediate simple instruction code. This set of instruction codes makes up the code part of the so called 'master object' held in computer memory. When you run an LPC program, the instruction set is traced, line by line as described above, causing the computer to execute a predefined set of actions defined by the instructions.

The difference between having interpreted and compiled code is that while the compiled code is quicker, the interpreted version is much esier to modify. If you want to change something in the compiled version you have to make the change in the source code, then recompile and store the new version, then try it out. With interpreted code you just make the change in the source and run it. With LPC you need to instruct the gamedriver to destroy the old master object instruction set as well, but more about that later.

Programs

LPC programs are described above as files containing instructions to the computer written in the language LPC. These files must be named something ending in the letters `.c' (e.g. `test.c') so that the gamedriver knows what it's dealing with; an LPC program. Program names can be any string of printable characters < 32 characters in length, beginning with an alphabetical letter. However, in practice it is recommendable that you limit yourself to < 16 letter strings that are made up of ordinary lowercase alphabetical letters only. If you want the name to be made up by two words, separate them with the `_'− character (e.g. `my_file_name.c').

Objects

An ***object*** in LPC is simply a copy of an existing and loaded program in computer memory. When a program is loaded into memory to produce a ***master object***, the code is compiled to produce the instruction list described earlier and a chunk of memory is associated to it as specified by the program code for use for internal ***global variables*** (described later). When a copy, a ***clone*** of this program is made, a special reference called an ***object pointer*** is created. That pointer is given a reference to the master code instruction list and a unique chunk of memory. If you clone the object another time, a new pointer is created and a new chunk of memory allocated. When an object is destroyed its associated memory is freed for use by other objects, but the instruction list is kept untouched. An object is ***always*** cloned from the master object. If you want to change the program you must ***update*** the master object to instruct the gamedriver that a new list of instructions is to be compiled from the source code.

However, any already existing clones will not change just because the master does. They will keep their reference to the old instruction list. It's important that you remember this so that you don't believe that the behaviour of an old cloned object changes just because you have updated the master object. As you see it's possible to have clones of objects in the game that behave differently, simply because they are made out of different source codes, just cloned between updates and changes in code. This could be a great source of confusion, so keep it in mind.

Object makeup

An object is comprised of something called ***functions*** and ***variables***. A function is a set of instructions you can reference by a name. A variable is a kind of container you can store data in for use by the functions. Some functions are already defined by the gamedriver, they are called ***external functions***, or ***efuns***. Functions defined in LPC code are called ***local functions***, or ***lfuns***. To confuse matters further there is a set of functions that count as efuns, but are written in LPC. These functions are called ***simulated efuns*** or ***sfuns***.

An efun basically is a function that is impossible to create in LPC. Take for example the function `write()` that allows you to present text on the screen of a player. It's impossible to make that up from other functions in LPC, so it has to be in the gamedriver. This efun is available in *all* LPC programs. Efuns also have no idea about what environment they are used in. They don't care one bit if they are used to simulate strawberry tasting, or as part of a game.

A function like `add_exit()` on the other hand, that adds an exit in a room is only available in room type objects. It is written in LPC. The lfuns as a rule are part of the makeup of the environment in which the objects are used. The example `add_exit()` for instance is well aware of such ideas as directions and travel costs, a very limited and specific concept.

The function `creator()` is a good example of the third case. It's a function that is available in every object, it returns information about who created a certain object. This information is *very* specific to the environment since it deals with such notions as code localization. This kind of function is easy to write in LPC but on the other hand it must be available in all objects, as if it was an efun. Due to this fact the special object `` `/secure/simul_efun.c' `` is made to be automatically available from all other objects in the game, you'll find all sfuns in there. This functionality is perfectly transparent to you; you just use them as you use any other efun and you don't have to be aware of that it really is an sfun.

Basic LPC

LPC if fairly similar to the language C, although some differences exist. As the experienced coder will find, it basically is a bit simplified with some new convenient types added and a set of functions to handle those types. Some inconsistancies exist but they are not serious enough to cause any problems as long as you are aware of them.

Comments

It might sound strange that I start off with this, but there'll be comments everywhere, so you need to be able to recognize them from the very start.

There are two kinds of comments:

```
<code> // This is a comment stretching to the end of the line.

<code> /* This is an enclosed comment */ <code>
```

As you can see, the first type of comment starts off with the `//` characters and then stretches all the way to the end of the line. If you want more lines of comments, you'll have to start off those as well with new `//` characters.

The second type is a type that has a definite length. They start with `/*` and end with `*/`. This kind of comment is useful when you have to write something that will stretch over several lines, as you only have to write the comment symbol in the start and the beginning.

NB! The `/* */` comment can *not* be nested. I.e. you can *not* write something like this for example:

```
/* A comment /* A nested comment */ the first continues */
```

What will happen is that the comment will end with the first found `*/`, leaving the text `the first continues */` to be interpreted as if it was LPC code. Naturally this won't work and instead you'll get an error message.

Data types

The object holds information in ***variables***. As the name hints these are labeled containers that may hold information that varies from time to time. It processes information in ***functions*** that both use and return data of various kinds.

In principle only one kind of data type is needed, a sort of general container that would cover anything you wanted to do. In reality it's much preferred if you can distinguish between different types of information. This might seem only to add to your programming problems, but in fact it reduces the risk of faulty code and improves legibility. It *much* improves on the time it takes to code and debug an object.

In LPC it is possible to use only that 'general purpose' data type I was talking about before. In the first versions of the language it was the only kind available. However, with the LPC we have today it is much preferrable if you avoid that as much as you can. In fact, start all your programs with the following instruction on a single line:

```
#pragma strict_types
```

This is an instruction to the gamedriver to check all functions so that they conform to the situation they are used in, and cause compile errors otherwise. This is a very great help in detecting programming errors early so that you don't wonder what's going on later when things don't quite turn out the way you wanted.

Now, the following types of data types are defined:

`*void*'
: `Nothing' This data type is used exclusively for functions that don't return any data at all.

`*int*'
: `Integers' Whole numbers in the range −2147483648 to 2147483647. e.g. `3, 17, −32, 999`.

`*float*'
: `Floating point numbers' Decimal numbers of any kind in the approximate range 1.17549435e−38 to 3.40282347e+38. e.g. `1.3, −348.4, 4.53e+4`. The range values are approximate since this might vary from mud to mud as it's platform dependant. If there are any FORTRAN fossiles around there, beware that numbers like `1.` or `.4711` are *not* recognized as floats, you have to specify both an integer and a decimal part, even if they only are 0.

`*string*'
: `Character strings' Strings are simply a series of printable characters within quotes, e.g. `"x", "the string", "Another long string with the number 5 in it"`. Strings can contain special characters like newline (`"\n"`) to end a line. A lot of LPC expressions can handle strings directly, unlike usual C. This makes strings very handy and easy to use.

`*mapping*'
: `Associated list' Mappings are another handy LPC invention (memory expensive, use with care!). A mapping is simply a list of associated values. Assume you want to remember the ages of people, like Olle is 23, Peter is 54 and Anna is 15. In LPC you can write this as `(["Olle":23, "Peter":54, "Anna":15])`. As you can see the value to the right has been associated to the value to the left. You can then extract the associated value through a simple indexing operation using the left hand value as index.

`*object*'
: `Object pointer' They are references to LPC programs that has been loaded into memory.

`*function*'
: `Function pointer' They are references to LPC functions.

`*Array*'
: All of the above can appear in arrays, indicated by a `*` in front of the variable name in the declaration. Arrays in LPC are more like lists than proper arrays. A number of functions and operator facilites exist to make them easy and quick to use.

`*mixed*'
: This, finally, is a general descriptor covering all the other, a sort of jack−of−all−trades type. Again, let me stress the fact that using it except when *absolutely* necessary only provokes mistakes. Hm, as pointed out to me this might sound a bit too strict. The mixed type is used for good reasons fairly often. What I mean is that when a regular type can be used, it should. Don't substitute it for mixed just because you feel lazy.

Variable declarations

A variable is a string of letters identifying an information 'box', a place to store data. The box is given a name consisting of < 32 characers, starting with an alphabetic letter. Custom and common sense dictate that all variables used inside a function consist of lowercase letters only. Global variables have the first letter in uppercase, the rest lowercase. No special character other than the '_' used to separate words is ever used. Variables should *always* be given names that reflect on their use. You declare variables like this:

```
<data type> <variable name>, <another variable>, ..., <last variable>;
e.g.
    int         counter;
    float       height, weight;
    mapping     age_map;
```

Variables must be declared at the beginning of a block (right after the first '{') and before any code statements. Global variables, variables that are available in all functions througout the program, should be declared at the top of the file.

When the declarations are executed as the program runs, they are initially set to 0, ***NOT*** to their 'null−state' values. In other words for example mappings, arrays and strings will all be set to 0 and not to `([])`, `({})` and `""` as you might believe. It is possible to initialize variables in the declaration statement, and it's even a very good habit always to initialize arrays and mappings there:

```
<data type> <variable name> = <value>, etc.
e.g.
    int         counter = 8;
    float       height = 3.0, weight = 1.2;
    mapping     age_map = ([]);
    object      *monsters = ({});
```

The reason why arrays and mappings should be initalized in the declaration statement to their 'NULL' values (`({})` and `([])` respectively) is that otherwise they are initialized to 0, which is incompatible with the proper type of the variable and might cause problems later.

Function declarations

A function must give proper notification of what kind of data type it returns, if any. A function is a label much like a variable name, consisting of < 32 characters, starting with a letter. Custom and common sense dictate that all function names should be lowercase and only contain the special character '_' to separate words. Use function names that clearly reflect on what they do. A function declaration looks like this:

```
/*
 * Function name: <Function name>
 * Description:   <What it does >
 * Arguments:
 * Returns:       <What the function returns>
 */
<return type>
<function name>(<argument list>)
{
   <code expressions>
}

/*
 * Function name: compute_diam
 * Description:   Compute the diameter of a circle given the
 *                circumference.
 * Variables:     surf_area − the surface area
 *                name − the name given the circle
 * Returns:       The circumference.
 */
float
compute_diam(float surf_area, string name)
{
    float rval;

    // Circumference = pie * diameter
    rval = surf_area / 3.141592643;
    write("The diameter of " + name + " is " + ftoa(rval) + "\n");

    return rval;
```

```
}
```

The argument list is a comma−separated list of data types, much like a variable declaration where you specify what kind of data will be sent to the function and assign names to this data for later use in the function. The data recieved will ***only*** be usable inside the function, unless you explicitly send it out through a function call.

(In order to save space and improve on legibility in the manual I won't put a header to all my short example functions).

A function that doesn't return anything should be declared as `void`.

```
void
write_all(string mess)
{
    users()->catch_msg(mess);
}
```

Statements and Expressions

We need to define what a ***statement*** and what an ***expression*** is in order to be able to proceed.

- Statements: Definition of statements
- Expressions: Definition of expressions
- Block statement: The block statement

Statements

A statement is sort of a full sentence of instructions, made up from one or more expressions. Statements usually cover no more than a single line of code. Sometimes it's necessary to break it up though if it becomes too long, simply to improve on legibility. For most statements you simply break the line between two words, but if you are in the middle of a string you need to add a backslash (\) at the end of the line in order for the gamedriver to understand what's going on.

```
write("This is an example of \
      a broken string.\n");
```

However, breaking a statement with backslash is extremely ugly and makes the code hard to read. In fact, it's usually possible to break the line naturally at the end of a string, between two operators of some kind, or even just split the string in half and add the two parts together with the + operator. The only time the backslash really is necessary is in `#define`−statements, handled later.

```
write("This is a better way of " +
      "breaking a string.\n");
```

Statements in LPC are usually ended with a `;`, which also is a good place to end the line. There's nothing stopping you from entering another statement right after, other than that it will look awful.

Expressions

An expression is an instruction or set of instructions that results in a value of some kind. Take +, for example. It uses two other expressions to make up a result. A variable is an expression since it yields its contents as a

result. The combination of the following two expressions and an operator is a valid expression: `a + b`, `a` and `b` being variables (expressions) and `+` being the operator used on them. `a = b + c;` is a full statement ending in a `;`.

Function calls are valid expressions. They are written simply as the name followed by a set of matched parentheses with the arguments that the functions uses listed inside. Take the simple function `max()` for example, that returns the max of the two arguments. To determine the maximum of `4` and `10`, you would write `max(4, 10)` as the expression. Naturally the result must be either stored or used.

The block statement

There are a lot of statements, for example conditional statements, that in certain circumstances execute *one* specified statement and never else. Suppose you want to have several statements executed and not just one? Well, there's a special statement called ***block*** statement that will allow you to do that. A block is defined as starting with a `{` and ending with a `}`. Within that block you may have as many statements of any kind (including variable definitions) as you like. The block statement is not ending with a `;`, even though it doesn't matter if you accidentally put one there.

The ';' statement

As stated `;` is mostly used to terminate statements, however it's also a statement in its own right.

The `;` on it's own will simply be a null−statement causing nothing to happen. This is useful when you have test−clauses and loops (described later) that perform their intended purpose within the test or loop clause and aren't actually intended to do anything else.

Scope and prototypes

Scope is a term defining where a function or variable declaration is valid. Since programs are read top down, left right (just like you read this page), declarations of functions and variables are available to the right and below of the actual declaration. However, the scope might be limited.

A variable that is declared inside a function is only valid until the block terminator (the terminating `}`) for that variable is reached.

```
< top of file >
int GlobCount;

// Only GlobCount is available here

void
var_func(int arg)
{
    int var_1;

    // GlobCount, arg and var_1 is available here
    < code >

    {
        string var_2;

        // GlobCount, arg, var_1 and var_2 is available in this block
        < code >
```

```
    }

    // GlobCount, arg and var_1 is available here
    < code >

    {
        int var_2;
        mapping var_3;

        // GlobCount, arg, var_1, var_2 and var_3 is available here
        // NB this var_2 is a NEW var_2, declared here
        < code >
    }

    // GlobCount, arg and var_1 is available here
    < code >
}

// Here only GlobCount (and the function var_func) is available
```

Function declarations follow the same rule, though you can't declare a function inside another function. However, suppose you have these two functions where the first uses the second:

```
int
func_1()
{
    < code >
    func_2("test");
}

void
func_2(string data)
{
    < code >
}
```

Then you have a problem, because the first function tries to use the second before it is declared. This will result in an error message if you have instructed the gamedriver to require types to match by specifying `pragma strict_types` as suggested earlier. To take care of this you can either re−arrange the functions so that `func_2` comes before `func_1` in the listing, but this might not always be possible and the layout might suffer. Better then is to write a ***function prototype***. The function prototype should be placed in the top of the file after the `inherit` and `#include` statements (described later) but *before* any code and look *exactly* as the function declaration itself. In this case:

```
< top of file, inherit and #include statements >

void func_2(string data);

< the actual code >
```

Operator expressions

The LPC language defines a large set of operators expressions, simply expressions that operate on other expressions. What follows here is a list of them. I've used a condensed notation so that the text won't take all of the page before getting down to actual explanations.

`E'

Any expression, even a compound one.

`V'

A variable.

- Miscellaneous operators: General (C) operators
- Arithmetic operators: The arithmetic operators
- Boolean operators: Boolean, or binary, operators
- Conditional operators: Conditionals
- Comparative operators: Comparing data

Miscellaneous operators

1. (E) E is evaluated before doing anything outside the parenthesis. This is useful for isolating expressions that need to be done in a specific order, or when you are uncertain about prescedence (described later).
2. E1, E2 E1 is evaluated first and the result stored, then E2 is evaluated and the result thrown away, lastly the stored result of E1 is returned as the value of the entire expression.

   ```
   The statement 'a = 1, 2, 3;' will set 'a' to contain '1'.
   ```

3. V = E The variable is given the value of the expression. The result of this entire expression is also the value of E.

   ```
   'a = b = 4;' will set a and b to be 4. It can also be written
   'a = (b = 4)' to illustrate the order of execution.
   ```

Arithmetic operators

1. E1 + E2 The expressions are evaluated and the results added to each other. You can add integers, floats, strings, arrays and mappings. Strings, arrays and mappings are simply concatenated − pasted together to the end of the first argument. It's also possible to add integers to strings, they will then be converted to strings and pasted to the end of the string.
2. E1 − E2 E2 is subtracted from E1. You can subtract integers, floats and any type from arrays of the same type. For arrays the item, if it exists in the array it is subtracted from, is removed from the array. If it doesn't exist in the array, the array is returned intact.
3. E1 * E2 E1 is multiplied by E2. This only works on integers and floats.
4. E1 / E2 E1 is divided by E2. This only works on integers and floats.
5. E1 % E2 The remainder of the expression 'E1 / E2' is returned. This only works with integers.

   ```
   '14 % 3' will yield 2 as the remainder. '14 / 3' will be 4, and
   4 * 3 + 2 = 14 as a small check.
   ```

6. −E Return E with reversed sign. This only works on integers and floats.
7. E++, ++E The expression 'E' is incremented by one. If the operator is in front of the expression, the incrememntation is done before the expression is used, otherwise afterwards.

   ```
   'a = 3; b = ++a;' will yield the result 'a = 4, b = 4', while
   'a = 3; b = a++;' will yield the result 'a = 4, b = 3'.
   ```

   This only works on integers.
8. E−−, −−E The expression 'E' is decremented by one. If the operator is in front of the expression, the deccrememntation is done before the expression is used, otherwise afterwards.

```
'a = 3; b = --a;' will yield the result 'a = 2, b = 2', while
'a = 3; b = a--;' will yield the result 'a = 2, b = 3'.
```

This only works on integers.

Boolean operators

Boolean (binary) operators are applicable only to integers with the exception of the & operator which also works on arrays. Internally an integer is 32 bits long. However, in the following examples I will only show the ten last bits as the others are 0 and can be ignored with the one exception of the ~−operator.

1. E1 & E2 E1 and E2.

   ```
   1011101001   (= 745)
   1000100010 & (= 546)
   ------------
   1000100000   (= 544) => 745 & 546 = 544
   ```

   Used on two arrays, this function will return a new array that holds all elements that are members of both of the argument arrays.
2. E1 | E2 E1 or E2.

   ```
   1011101001   (= 745)
   1000100010 | (= 546)
   ------------
   1011101011   (= 747) => 745 | 546 = 747
   ```

3. E1 ^ E2 E1 xor (exclusive or) E2.

   ```
   1011101001   (= 745)
   1000100010 ^ (= 546)
   ------------
   0011001011   (= 203) => 745 ^ 546 = 203
   ```

4. ~E 1−complement of E (invert E).

   ```
   00000000000000000000001011101001 ~ (= 745)
   --------------------------------
   11111111111111111111110100010110   (= -746) => ~745 = -746
   ```

   NB! The above example might be hard to understand unless you really know your binary arithmetic. However, trust me when I say that this is *not* a typo, it's the way it should look. Read a book on boolean algebra (the section on two−complement binary arithmentic) and all will be clear.
5. E1 << E2 E1 is shifted left E2 steps.

   ```
   5 << 4 => 101(b) << 4 = 1010000(b) = 80
   ```

6. E1 >> E2 E1 is shifted right E2 steps.

   ```
   1054 >> 5 => 10000011110(b) >> 5 = 100000(b) = 32
   ```

Conditional (logical) operators

1. E1 || E2 Returns true if E1 or E2 evaluates as true. Will not evaluate E2 if E1 is true.
2. E1 && E2 Returns true if both E1 and E2 evaluates as true. Will not evaluate E2 if E1 is false.
3. !E Returns true if E is false & vice versa.

Comparative operators

1. E1 == E2 Returns true if E1 is equal to E2, can be used on all kinds of types, but see the special section later on arrays and mappings, it works differently on them from what you might think.
2. E1 != E2 Returns true if E1 isn't equal to E2, can be used on all kinds of types, but see the special section later on arrays and mappings, it works differently om them from what you might think.
3. E1 > E2 Returns true if E1 is greater than E2, can be used on all types except arrays and mappings.
4. E1 < E2 Returns true if E1 is less than E2, can be used on all types except arrays and mappings.
5. E1 >= E2 Returns true if E1 is greater or equal to E2, can be used on all types except arrays and mappings.
6. E1 <= E2 Returns true if E1 is less or equal to E2, can be used on all types except arrays and mappings.

Prefix allocation

All of the arithmetic and boolean operator expressions can be written in a shorter way if all you want to do is compute one variable with any other expression and then store the result in the variable again.

Say that what you want to do is this `a = a + 5;`, a much neater way of writing that is `a += 5;`. The value of the second expression is added to the first and then stored in the first which happens to be a variable.

You write all the others in the same way, i.e. the variable, then the operator directly followed by = and then the expression.

```
a >>= 5;        // a = a >> 5;
b %= d + 4;     // b = b % (d + 4);
c ^= 44 & q;    // c = c ^ (44 & q);
```

Precedence and Order of evalutaion

The table below summarizes the rules for precedence and associativity of all operators, including those which we have not yet discussed. Operators on the same line have the same precedence, rows are in order of decreasing precedence, so, for example, `*`, `/` and `%` all have the same precedence, which is higher than that of `+` and `−`.

Note that the precedence of the bitwise logical operators `&`, `^` and `|` falls below `==` and `!=`. This implies that bit−testing expressions like

```
if ((x & MASK) == 0) ...
```

must be fully parenthesized to give proper results.

1. () [] Left to right
2. ! ~ ++ −− − (type) * & Right to left

3. * / % Left to right
4. + − Left to right
5. << >> Left to right
6. < <= > >= Left to right
7. == != Left to right
8. & Left to right
9. ^ Left to right
10. | Left to right
11. && Left to right
12. || Left to right
13. ?: Right to left
14. = += == etc. Right to left
15. , Left to right

Conditionals

Conditional statements are used a lot in LPC, and there is several ways of writing them. A very important concept is that 0 is considered as ***false*** and any other value as ***true*** in tests. This means that empty listings `({})`, empty strings `""` and empty mappings `([])` also are evaluated as ***true*** since they aren't `0`. You have to use special functions to compute their size or determine content if you want test them, more about that later however.

- if/else: The common if/else statement
- switch: The switch statement
- ?colon: The ?: statement

The if/else statement

The most common conditional statement is naturally the `if` statement. It's easy to use and can be combined with an `else` clause to handle failed tests. It's written like this:

```
if (expression) statement;
e.g.
    if (a == 5)
        a -= 4;
```

If you want to handle the failed match, you add an else statement like this:

```
if (expression) true-statement else false-statement;
e.g.
    if (a == 5)
        a -= 4;
    else
        a += 18;
```

The switch statement

If one variable has to be tested for a lot of different values, the resulting list of `` `if-else-if-else' `` statements soon gets very long and not very easy to read. However, if the type of the value you are testing is an integer, a float or a string you can use a much denser and neater way of coding. Assume you have the

following code you want to write:

```
if (name == "fatty")
{
    nat = "se";
    desc = "blimp";
}
else if (name == "plugh")
{
    nat = "no";
    desc = "warlock";
}
else if (name == "olorin")
{
    nat = "de";
    desc = "bloodshot";
}
else
{
    nat = "x";
    desc = "unknown";
}
```

The better way of writing this is as follows:

```
switch (name)
{
case "fatty":
    nat = "se";
    desc = "blimp";
    break;

case "plugh":
    nat = "no";
    desc = "warlock";
    break;

case "olorin":
    nat = "de";
    desc = "bloodshot";
    break;

default:
    nat = "x";
    desc = "unknown";
}
```

The workings of this statement is very simple really: `switch` sets up the expression value within the parenthesis for matching. Then every expression following a `case` is examined to find a match.

NB! The `case` expression ***must*** be a constant value, it can't be a variable, function call or other type of expression.

After a match has been found the following statements are executed until a `break` statement is found. If no matching value can be found, the `default` statements are executed instead.

NB! While it's not mandatory to have a `default` section, it's highly recommended since that usually means that something has happened that wasn't predicted when writing the program. If you have written it that way

on purpose that's one thing, but if you expect only a certain range of values and another one turns up it's usually very good to have an error message there to notify the user that something unexpected happened.

If you forget to put in a 'break' statement the following 'case' expression will be executed. This might sound like something you don't want, but if in the example above the names `fatty' and `plugh' both should generate the same result you could write:

```
case "fatty":
    /* FALLTHROUGH */
case "plugh":
    < code >
    break;
```

... and save a bit of space. Writing code with switch doesn't make it any quicker to execute, but a lot easier to read thereby reducing the chance of making mistakes while coding. Remember to put the `/* FALLTHROUGH */` comment there though, or it might be hard to remember later if it was intentional or an ommission of a `break` statement, particularly if you have some code that's executed previously to the fallthrough. A good idea is usually to add an extra linefeed after a `break` statement just to give some extra 'breathing space' to improve on legibility.

The ?: expression

This is a very condensed way of writing an `if/else` statement and return a value depending on how the test turned out. This isn't a statement naturally, it's an expression since it returns a value, but it was hard to explain earlier before explaining the if/else statement.

Suppose you want to write the following:

```
if (test_expression)
    var = if_expression;
else
    var = else_expression;
```

You can write that much more condensed in this way:

```
var = test_expression ? if_expression : else_expression;
e.g.
    name = day == 2 ? "tuesday" : "another day";
```

It can be debated if writing code this way makes you code easier or harder to read. As a rule it can be argued rather successfully that one expression of that kind does make it clearer, but that a combination of several only makes it worse. Something like this *definately* isn't an improvement:

```
name = day == 2 ? time == 18 ? "miller time" : "tuesday" : "another day";
```

Loop statements

Basically there are two kinds of loop statements which incorporate the use of conditional statements within them, i.e. they can be programmed to execute only until a certain state is reached.

- for: The for statement
- while: The while statement

The for statement

If you want a simple counter you should use the ***for*** statement. The syntax is as follows:

```
for (initalize_statement ; test_expression ; end_of_loop_statement)
    body_statement;
```

When first entered, the `for` statement executes the ***initialize_statement*** part. This part usually is used to initialize counters or values used during the actual loop. Then the actual loop starts. Every loop starts by executing the ***test_expression*** and examining the result. This is a truth conditional, so any answer not equal to 0 will cause the loop to be run. If the answer is true the ***body_statement*** is executed, immediately followed by the ***end_of_loop_statement***. In the ***body_statement*** you usually do what you want to have done during the loop, in the ***end_of_loop_statement*** you usually increment or decrement counters as needed.

Throughout the previous section I used the word *usually* a lot. This is because you don't *have* to do it that way, there's no rule forcing you to make use of the statements in the way I said. However, for now let's stick to the regular way of using the ***for−statement***. Later on I'll describe more refined techniques.

Assume you want to compute the sum of all integers from 7 to 123 and don't know the formula ((x2^2 + x1^2) / 2). The easiest (if not most efficient) way of doing that is a loop.

```
result = 0;
for (count = 7 ; count < 124 ; count++)
    result += count;
```

What happens is that first of all result is set to 0, then the actual `for`−statement is entered. It begins by setting the variable count to 7. Then the loop is entered, beginning by testing if count (= 7) is less than 124, it is so the value is added to count. Then count is incremented one step and the loop entered again. This goes on until the count value reaches 124. Since that isn't less than 124 the loop is ended.

NB! The value of count after the `for`−statement will be 124 and *not* 123 that some people tend to believe. The ***test_expression*** must evaluate to `false` in order for the loop to end, and in this case the value for count then must be 124.

The while statement

The ***while*** statement is pretty straight−forward, you can guess from the very name what it does. The statement will perform another statement over and over until a given `while` expression returns false. The syntax is simple:

```
while (<test expression>)
```

Note carefully that the test expression is checked first of all, before running the statement the first time. If it evaluates as false the first time, the body is never executed.

```
a = 0;
while (a != 4)
{
    a += 5;
    a /= 2;
}
```

The break and continue statement

Sometimes during the execution of `switch`, `for` or `while` statements it becomes necessary to abort execution of the block code, and continue execution outside. To do that you use the `break` statement. It simply aborts execution of that block and continues outside it.

```
while (end_condition < 9999)
{
    // If the time() function returns 29449494, abort execution
    if (time() == 29449494)
        break;

    < code >
}

// Continue here both after a break or when the full loop is done.
< code >
```

Somtimes you merely want to start over from the top of the loop you are running, in a `for` or `while` statement, that's when you use the `continue` statement.

```
// Add all even numbers
sum = 0;
for (i = 0 ; i < 10000 ; i++)
{
    // Start from the top of the loop if 'i' is an odd number
    if (i % 2)
         continue;

    sum += i;
}
```

Arrays and Mappings

It's time to dig deeper into the special type ***array*** and ***mapping***. Their use might look similar but in fact they are very different from each other as you will see.

To both of these data types there exists a number of useful (indeed even essential) efuns that manipulates them and extracts information from them. They will be described in total later however, only some are mentioned here.

- Arrays: How to declare and use arrays
- Mappings: How to declare and use mappings

How to declare and use arrays

Arrays really aren't arrays in the proper sense of the word. They can better be seen as lists with fixed order. The difference might seem slight, but it makes sense to the computer−science buffs :)

Arrays are type−specific. This means that an array of a certain type only can contain variables of that single type. Another restrictions is that all arrays are one−dimensional. You can't have an array of arrays. However, the `mixed` type takes care of these limitations. A mixed variable can act as an array containing any data type,

even other arrays. As a rule you should try to use properly typed arrays to minimize the probabilities of programming mistakes however.

You declare an array like this:

```
<type> *<array name>;
e.g.
    int *my_arr, *your_arr;
    float *another_arr;
    object *ob_arr;
```

The initial values of these declared arrays is '0', *not* an empty array. I repeat: they are initialized to 0 and *not* to an empty array. Keep this in mind!

You can allocate and initialize an array like this:

```
<array> = ({ elem1, elem2, elem3, ..., elemN });
e.g.
    my_arr = ({ 1, 383, 5, 391, -4, 6 });
```

You access members of the array using brackets on the variable name. (Assume *val* here is declared to be an integer).

```
<data variable> = <array>[<index>];
e.g.
    val = my_arr[3];
```

LPC, like C, starts counting from 0, making the index to the fourth value = 3.

To set the value of an existing position to a new value, simply set it using the = operator.

```
    my_arr[3] = 22;     // => ({ 1, 383, 5, 22, -4, 6 })
    my_arr[3] = 391;    // => ({ 1, 383, 5, 391, -4, 6 })
```

If you want to make a subset of an array you can specify a range of indices within the brackets.

```
<array variable> = <array>[<start_range>..<end_range>];
e.g.
    your_arr = my_arr[1..3];
```

... will result in `your_arr` becoming the new array `({ 383, 5, 391 });` If you give a new value to an old array, the previous array is lost.

```
e.g.
    my_arr = ({ });
```

... will result in `my_arr` holding an empty array. The old array is deallocated and the memory previously used is reclaimed by the gamedriver.

If you index outside an array, an error occurs and execution of the object is aborted. However, range indexing outside the array does not result in an error, the range is then only constrained to fall within the array.

If you want to create an empty array, initialized to 0 (no matter the type of the array, all positions will be set to 0 anyway) of a given length, you use the efun `allocate()`.

```
<array> = allocate(<length>);
e.g.
    your_arr = allocate(3);     // => your_arr = ({ 0, 0, 0 });
```

Concatenating (adding) arrays to each other is most easily done with the + operator. Simply add them as you would numbers. The += operator works fine as well.

```
my_arr = ({ 9, 3 }) + ({ 5, 10, 3 }); // => ({ 9, 3, 5, 10, 3 })
```

Removing elements from an array is easiest done with the −/−= operator, however, be aware that it is a general operator that will remove *all* items found that match the item you want to remove.

```
my_arr −= ({ 3, 10 }); // => ({ 9, 5 })
```

If you want to remove a single item in the middle somewhere that might have been repeated, you have to use the range operator of course.

```
my_arr = ({ 9, 3, 5, 10, 3 });
my_arr = my_arr[0..0] + my_arr[2..4]; // => ({ 9, 5, 10, 3 })
```

NB! Beware this difference!!!! One is a list, the other an integer!

```
    <array> my_arr[0..0]   // = ({ 9 })
    <int>   my_arr[0]      // = 9
```

How to declare and use Mappings

Mappings are lists of associated values. They are `mixed` by default, meaning that the index part of the associated values doesn't have to be of the same type all the time, even though this is encouraged for the same reason as before in regard to the mixed data type.

Mappings can use any kind of data type both as index and value. The index part of the mapping in a single mapping must consist of unique values. There can *not* be two indices of the same value.

This all sounds pretty complicated, but in reality it's pretty simple to use. However, it will be a lot easier to understand once we get down to actually seeing it used.

You declare a mapping just like any other variable, so let's just start up with a few declarations for later use:

```
mapping my_map;
int     value;
```

Allocating and initializing can be done in three different ways:

```
1:      <mapping_var> = ([ <index1>:<value1>, <index2>:<value2>, ... ]);

2:      <mapping_var>[<index>] = value;

3:      <mapping_var> = mkmapping(<list of indices>, <list of values>);
```

The first is straight−forward and easy.

```
1: my_map = ([ "adam":5, "bertil":8, "cecar":−4 ]);
```

The second works so that in case a given mapping pair doesn't exist, it is created when referenced. If it does exist the value part is replaced.

```
2: my_map["adam"] = 1;    // Creates the pair "adam":1
   my_map["bertil"] = 8;  // Creates the pair "bertil":8
   my_map["adam"] = 5;    // Replaces the old value in "adam" with 5.
        ...
```

The third requires two arrays, one containing the indices and one containing the values. How to create arrays was described in the previous chapter.

```
3: my_map = mkmapping(({ "adam", "bertil", "cecar" }), ({ 5, 8, -4 }));
```

Unlike arrays there's no order in a mapping. The values are stashed in a way that makes finding the values as quick as possible. There are functions that will allow you to get the component lists (the indices or values) from a mapping but keep in mind that they can be in *any* order and are not guaranteed to remain the same from call to call. In practice though, they only change order when you add or remove an element.

Merging mappings can be done with the `+/+=` operator just as with mappings.

```
my_map += ([ "david":5, "erik":33 ]);
```

Removing items in a mapping, however, is a bit trickier. That has to be done by using the special efun `m_delete()` (also described later).

```
my_map = m_delete(my_map, "bertil");
my_map = m_delete(my_map, "david");
```

As you see the mapping pairs has to be removed one by one using the index as an identifier of which pair you want to remove. Another thing you now realize quite clearly is that the indices in a mapping has to be unique, you can't have two identical 'handles' to different values. The values however can naturally be identical.

Individual values can be obtained through simple indexing.

```
value = my_map["cecar"]; // => -4
```

Indexing a value that doesn't exist will *not* generate an error, only the value 0. Be *very* careful of this since you might indeed have legal values of 0 in the mapping as well. i.e. a value of 0 might mean that the index has no value part but also that the value indeed *is* 0.

```
value = my_map["urk"]; // => 0
```

The preprocessor

The preprocessor is *not* a part of the LPC language proper. It's a special process that is run before the actual compilation of the program occurs. Basically it can be seen as a very smart string substitutor; Specified strings in the code is replaced by other strings.

All preprocessor directives are given as strings starting with the character `#' on the first column of the line. You can put them anywhere, but as you'll be told later most of them do belong in the beginning of the code.

- #include: The #include statement
- #define: The #define and undef statement
- #if etc: The #if, #ifdef, #ifndef, #else and #elseif statements

The #include statement

This is by far the most used preprocessor command. It simply tells the preprocessor to replace that line with the contents of an *entire* other file before going any further.

Data you put in ***include−files*** is usually data that won't ever change and that you want to put into several files. Instead of having to write those same lines over and over with the cumulative chance of putting in copying errors as you go, you simply collect that data into one or more files and include them into the program files as necessary.

The syntax is very easy:

```
#include <standard_file>
#include "special_file"
```

NB! Note the absence of a `;` after the line!

The two different ways you write this depend on where the file you want to include exists. There's a number of standard include files in the game, spread around in a number of different directories. Rather than having to remember exactly where they are, you can just give the name of the file you want to include then.

```
#include <stdproperties.h>
#include <adverbs.h>
```

If you want to include files that aren't part of the standard setup, for example files of your own, you have to specify where they are. You do that either relative to the position of the file that uses it or by an absolute path.

```
#include "/d/Genesis/login/login.h"
#include "my_defs.h"
#include "/sys/adverbs.h"       // Same as the shorter one above
```

When you include standard files, *always* use the <>−path notation. The reason isn't only that it becomes shorter and easier to distingues but also that if the files move around your program will stop working. If you use the <>−notation they will always be found anyway.

Include files can have any name, but as a rule they are given the '.h' suffix to clearly distinguish them as include files.

It is even possible to include `c`−files, i.e. to include entire files full of code. However, doing that is `very` bad form. Do *not* do that ***EVER***! Why? Well, for one thing error handling usually has a bad time tracing errors in included files, the line numbers gets wrong. Also, since you include the uncompiled code into several different objets, you will waste memory and CPU since these identical included parts has to be compiled and stored separately for each object that uses them. Apart from all this just reading the code will be a chore better not even contemplated.

What has the extension of the file name *really* to do with the contents then? Well... actually nothing at all. However, the convention is to keep code, functions that are to be executed, in `c`−files and definitions in

h−files. Usually the mudlib reflects on this convention and might not recognize anything but c−files as code sources.

The #define statement

This is a very powerful ***macro*** or substitute preprocessor command that can be abused endlessly. You are wise if you use it with caution and only for simple tasks.

The syntax is as follows:

```
#define <pattern> <substitute pattern>
#undef <pattern>
```

Any text in the file that matches `<pattern>` will be substituted for `<substitute pattern>` before compilation occurs. A `#define` is valid from the line it is found on until the end of the file or an `#undef` command that removes it.

Although defines can be written just as any kind of text, it is the custom (do this!) to use only capitals when writing them. This is so that they will be easily distinguishable for what they are since no one (not you either!) ever writes function or variable names with capitals.

Place all defines in the beginning of the file, or the poor chum who next tries to look at your code will have the devil's own time of locating them. If it's someone you asked for help (since your badly written code most likely won't work) he probably will tell you to stick the file someplace very unhygienic and come back later when you've learned to write properly.

Simple defines are for example paths, names and above all constants of any kind that you don't want to write over and over.

```
#define MAX_LOGIN  100          /* Max logged on players */
#define LOGIN_OB   "/std/login" /* The login object      */
#define GREET_TEXT "Welcome!"   /* The login message     */
```

Wherever the pattern strings above occur, they will be replaced by whatever is followed by the pattern until the end of the line. That includes the comments above, but they are removed anyway later.

```
tell_object(player, GREET_TEXT + "\n");
```

A comment on the `//` form is *not* a good thing since it doesn't end until the end of the line.

```
#define GREET_TEXT    "Welcome!"    // The login message
```

...will be translated into the previous example as:

```
tell_object(player,    "Welcome!"    // The login message + "\n");
```

...which will have the effect of commenting away everything after the `//`, all the way until the end of the line.

If a macro extends beyond the end of the line you can terminate the lines with a \ which signifies that it continues on the next line. However, you *must* break the string right after the \, there must ***NOT*** be any spaces or other characters there, just the linebreak.

```
#define LONG_DEFINE  "beginning of string \
                      and end of the same"
```

Function−like defines are fairly common and often abused. The only really important rule is that any argument to the macro *must* be written so that they are used enclosed in parenthesis. If you don't do that you can end up with some very strange results.

```
1: #define MUL_IT(a, b) a * b        /* Wrong */
2: #define MUL_IT(a, b) (a * b)      /* Not enough */
3: #define MUL_IT(a, b) ((a) * (b))  /* Correct */
```

What's the big difference you may ask? Well, look at this example:

```
result = MUL_IT(2 + 3, 4 * 5) / 5;

   Expanded this becomes:

1: result = 2 + 3 * 4 * 5 / 5;       // = 14, Wrong
2: result = (2 + 3 * 4 * 5) / 5      // = 12, Just as wrong
3: result = ((2 + 3) * (4 * 5)) / 5  // = 20, Correct!
```

Abuse of defines usually involves badly formulated macros, complicated macros used inside other macros (making the code almost impossible to understand) or humungous arrays or mappings in defines that are used often. The basic rule is to keep macros short and fairly simple. Do that and you'll never have any problems.

The #if, #ifdef, #ifndef, #else and #elseif statements

These are all preprocessor directives aimed at selecting certain parts of code and removing other depending on the state of a preprocessor variable.

The `#if` statement looks very much like a normal if statement, just written a bit differently.

Assume you *may* have the following define somewhere:

```
#define CODE_VAR  2

or

#define CODE_VAR  3
```

Then you can write

```
#if CODE_VAR == 2
    <code that will be kept only if CODE_VAR == 2>
#else
    <code that will be kept only if CODE_VAR != 2>
#endif
```

You don't have to have the `#else` statement there at all if you don't want to.

It's sufficient to have the following statement to 'define' a preprocessor pattern as existing:

```
#define CODE_VAR    /* This defines the existance of CODE_VAR */
```

Then you can write like this:

```
#ifdef CODE_VAR
    <code that will be kept only if CODE_VAR is defined>
#else
    <code that will be kept only if CODE_VAR isn't defined>
#endif
```

or

```
#ifndef CODE_VAR
    <code that will be kept only if CODE_VAR isn't defined>
#else
    <code that will be kept only if CODE_VAR is defined>
#endif
```

Again, the `#else` part is optional.

The `#if/#ifdef/#ifndef` preprocessor commands are almost only used to add debug code that you don't want to have activated all of the time, or code that will work differently depending on other very rarely changing parameters. Since the conditions have to be hard−coded in the file and can't change during the course of the use of the object this is something you very rarely do.

Essential LPC and Mudlib

This chapter will teach you what you need to know in order to actually produce code in the game environment. It will avoid the more complicated and unessential subjects, leaving them for chapter three. You will be taught a lot more about he mudlib and the workings of the gamedriver, knowledge necessary to produce working and effective code.

Peeking at things to come

In order to provide you with examples of what I'm trying to teach you, I need to explain a few functions in advance. They will be repeated in their correct context later, but here's a preview so that you'll know what I'm doing.

To present things on the screen for the player to read, you use the efun `write()`. There's two special characters that's often used to format the text, `` `tab' `` and `` `newline' ``. They are written as `\t` and `\n` respectively. The `` `tab' `` character inserts 8 space characters and `` `newline' `` breaks the line.

```
void write(string text)
e.g.
    write("Hello there!\n");
    write("\tThis is an indented string.\n");
    write("This is a string\non several lines\n\tpartly\nindented.\n");

    /* The result is:

       Hello there!

               This is an indented string.

       This is a string
       on several lines
               partly
       indented.
     */
```

If you have an array, mapping, or simply a variable of any kind that you want displayed on the screen for debugging purposes the sfun `dump_array()` is very handy. You simply give the variable you want displayed as an argument and the contents (any kind) will be displayed for you.

```
void dump_array(mixed data)
e.g.
    string *name = ({ "fatty", "fido", "relic" });
    dump_array(name);

    /* The result is

       (Array)
       [0] = (string) "fatty"
       [1] = (string) "fido"
       [2] = (string) "relic"
     */
```

LPC revisited

Let's start by ripping off the rest of the LPC that was avoided in the first chapter. We need this in order to be able to actually create working objects in the game environment.

- Function calls: Making function calls
- Inheritance: Inheriting object classes
- Shadows: Masking functions in runtime
- Type identification: Determining the type of a variable
- Type qualifiers: Further specify the functionality
- Function definition (part 2): The function type part 2
- switch (part 2): switch/case part 2
- catch/throw: Error handling during runtime
- Array & Mapping references: Reference by value or pointer

Function calls

There are two kinds of function calls, internal and external. The only kind we have discussed so far is the internal one even though the external call has been displayed a few times.

- Internal calls: Making object−internal function calls
- External single calls: Making object−external single function calls
- External multiple calls: Making object−external multiple function calls

Making object−internal function calls

[call_self]

Making an internal function call is as simple as writing the function name and putting any arguments within parentheses afterwards. The argument list is simply a list of expressions, or nothing. (A function call is naturally an expression as well).

```
<function>(<argument list>);
e.g.
    pie = atan(1.0) * 4;
```

There is another way of doing this as well. If you have the function name stored in a string and wish to call the function, you use the efun `call_self()`:

```
call_self(<"function name">, <argument list>);
e.g.
    pie = call_self("atan", 1.0) * 4;
```

If you use `call_self()` and specify a function that doesn't exist, you will get an error message and the execution of the object will be aborted.

Making single object−external function calls

[call_other]

An external call is a call from one object to another. In order to do that you need an object reference to the object you want to call. We haven't discussed exactly how you acquire an object reference yet, but assume for the moment that it already is done for you.

```
mixed <object reference/object path>->&lt;function&gt;(&lt;argument list&gt;);
mixed call_other(<ob ref/ob path>, "<function>", <arg list>);
e.g.
    /*
     * Assume that I want to call the function 'compute_pie' in the
     * object "/d/Mydom/thewiz/math_ob", and that I also have the
     * proper object pointer to it stored in the variable 'math_ob'
     */
    pie = math_ob->compute_pie(1.0);
    pie = "/d/Mydom/thewiz/math_ob"->compute_pie(1.0);
    pie = call_other(math_ob, "compute_pie", 1.0);
    pie = call_other("/d/Mydom/thewiz/math_ob", "compute_pie", 1.0);
```

As you can see, the efun `call_other()` works analogous to `call_self()`.

If you make an external call using the object path, the so called ***master object*** will be called. If the object you call hasn't been loaded into memory yet, it will be. If an external call is made to a function that doesn't exist in the object you call, 0 will be returned without any error messages. Calls to objects that have bugs in the code *will* result in an error message and the execution of the object that made the call is aborted.

What's the big deal with `call_self()` then? Why can't you just use `call_other()` all the time with the same result? Well, it has to do with access to functions in an object, a part of LPC that I haven't gotten around to explaining yet. However, the difference is that `call_self` really works just like any internal call in regard to function modifiers and function access, while `call_other()` is a pure external call. Remember this for the future when differences between internal and external access to functions are discussed.

Making multiple object-external function calls

You can call several objects at once just as easily as a single one. If you have an array of path strings or object pointers, or a mapping where the value part are path strings or object pointers you can call all the referenced objects in one statement. The result will be an array with the return values if you call using an array and a mapping with the same index values as the calling mapping if you give a mapping.

```
(array/mapping) <array/mapping>->&lt;function&gt;(&lt;argument list&gt;);
e.g.
    /*
     * I want a mapping where the indices are the names of the players
     * in the game, and the values are their hitpoints.
     */
    object *people, *names;
    mapping hp_map;

    // Get a list of all players.
    people = users();

    // Get their names.
    names = people->query_real_name();

    // Make a mapping to call with. Item = name:pointer
    hp_map = mkmapping(names, people)

    // Replace the pointers with hitpoint values.
    hp_map = hp_map->query_hp();
```

```
    // All this could also have been done simpler as:
    hp_map = mkmapping(users()->query_real_name(), users()->query_hp());
```

Inheriting object classes

Assume that you want to code an item like a door, for example. Doing that means that you have to create functionality that allows the opening and closing of a passage between two rooms. Perhaps you want to be able to lock and unlock the door, and perhaps you want the door to be transparent. All of this must be taken care of in your code. Furthermore, you have to copy the same code and make small variations in description and use *every time* you want to make a new door.

After a while you'll get rather tired of this, particularly as you'll find that other wizards has created doors of their own that work almost − but not quite − the same way your does, rendering nifty objects and features useless anywhere but in your domain.

The object oriented way of thinking is that instead of doing things over and over you create a basic door object that can do all the things you want any door to be able to do. Then you just inherit this generic door into a specialized door object where you configure exactly what *it* should be able to do from the list of available options in the ***parent*** door.

It is even possible to inherit several different objects where you can combine the functionality of several objects into one. However, be aware that if the objects you inherit define functions with the same names, they will indeed clash. Just be aware of what you are doing and why, and you won't have any problems.

The syntax for inheriting objects is very simple. In the top of the file you write this:

```
inherit "<file path>";
e.g.
        inherit "/std/door";
        inherit "/std/room.c";
```

NB! This is ***NOT*** a preprocessor command, it is a statement, so it does ***NOT*** have a # in front of it, and it is ended with a `;`. As you see you may specify that it's a c−file if you wish, but that's not necessary.

The ***child*** will inherit all functions and all variables that are declared in such a way as to permit inheriting. If you have a function with the same name as a function in the parent, your function will ***mask*** the parent one. When the function is called by an external call, your function will be executed. Internal calls in the parent will still go to the parent function. Often you need to call the parent function anyway from the child, you do that by adding `::` to the internal function call.

```
void
my_func()
{
    /*
     * This function exists in the parent, and I need to
     * call it from here.
     */
    ::my_func();        // Call my_func() in the parent.
}
```

It is not possible to call a masked function in the parent by an external call, it is only available from within the object itself. If an object inherits an object that has inherited another object, e.g. C inherits B that inherits A,

then masked functions in A is only available from B, not from C.

Shadows: Masking functions during runtime

There's a functionality called ***shadowing*** available in LPC. Purists tend to use the word 'abomination' and scream for its obliteration, since it goes against most of what's taught about proper control flow and object purity. For gaming purposes it's rather useful, although it can cause a host of problems (particularly when it comes to security). Use it with caution!

It's possible to make an object ***shadow*** another object. What happens then is that the functions and global variables in the shadow object that also exist in the shadowed object mask the original. Calls to the shadowed functions will go to the shadow instead. The shadow will for all practical appearances 'become' the object it shadows. As you can see this is done in runtime, and not during compilation.

This is all I will say in this respect right now. How to create shadows and use them will be handled in detail later. For now this is what you need to know.

Type identification

[intp, floatp, functionp, stringp, objectp, mappingp, pointerp]

Due to the fact that all variables are initialized to 0, and that many functions return 0 when failing, it's desirable to be able to determine what kind of value type you actually have received. Also, if you use the `mixed` type it's virtually essential to be able to test what the variable contains at times. For this purpose there's a special test function for each type that will return 1 (true) if the tested value is of the asked for type, and 0 if not.

@bullet{int intp(mixed)}
: Test if given value is an integer

@bullet{int floatp(mixed)}
: Test if given value is a float

@bullet{functionp(mixed)}
: Test if given value is a function pointer

@bullet{int stringp(mixed)}
: Test if given value is a string

@bullet{int objectp(mixed)}
: Test if given value is an object pointer

@bullet{int mappingp(mixed)}
: Test if given value is a mapping

@bullet{int pointerp(mixed)}
: Test if given value is an array

NB! These functions test the type of the value, ***NOT*** the value itself in the sense of truth functionality. In other words `intp(0)` will always evaluate as true, as will `mappingp(([]))`.

Type qualifiers

The very types you assign variables and function can have qualifiers changing the way they work. It's very important to keep them in mind and use the proper qualifier at the proper time. Most work differently when applied to variables rather than functions, so a bit of confusion about how they work usually is quite common

among the programmers. Try to get this straight now and you'll have no problems later

- static (variable): The static variable qualifier
- static (function): The static function qualifier
- private: The private function/variable qualifier
- nomask: The nomask function/variable qualifier
- public: The public function/variable qualifier
- varargs: The varargs function qualifier

The static variable qualifier

This is a problematic qualifier in the respect that it works differently even for variables depending on where they are! Global variables, to begin with, are (as you know) variables that are defined in the top of the file *outside* any function. These variables are available in all functions i.e. their ***scope*** is object−wide, not just limited to one function.

It is possible to save all global variables in an object with a special efun (described later). However, if the global variable is declared as `static`, it is *not* saved along with the rest.

```
static string   TempName;        // A non−saved global var.
```

The static function qualifier

A function that is declared `static` can not be called using external calls, only internal. This makes the function 'invisible' and inaccessable for other objects.

The private function/variable qualifier

A variable or function that has been declared as `private` will not be inherited down to another object. They can only be accessed within the object that defines it.

The nomask function/variable qualifier

Functions and variables that are declared as `nomask` can not be masked in any way, neither by shadowing nor inheriting. If you try you will be given an error message.

The public function/variable qualifier

This is the default qualifier for all functions. It means there is no limits other than those which the language imposes on accessing, saving and masking.

The varargs function qualifier

Sometimes you want a function to be able to receive a variable amount of arguments. There's two ways of doing this and it can be discussed if it's correct to put both explanations in this chapter, but it's sort of logical

to do so and not too hard to find.

A function that is defined `varargs` will be able to receive a variable amount of arguments. The variables that aren't specified at the call will be set to 0.

```
varargs void
myfun(int a, string str, float c);
{
}
```

The call `myfun(1);` will set `a` to `1`, `str` and `c` to `0`. Make sure you test the potentially unused variables before you try to use them so that they do contain a value you can use.

There's another way as well. You can specify default values to the variables that you're uncertain about. Then you don't have to declare the function `varargs` and you will have proper default values in the unused argument variables as well.

```
void
myfun(int a, string str = "pelle", float c = 3.0);
{
}
```

This function must be called with at least one argument, the first, as it wasn't given a default value. The call `myfun(1, "apa");` will set `a` to `1`, `str` to `"apa"` and `c` to `3.0`.

The function data type, part 2

There's one data type that I more or less ignored earlier, and that's the `function` type. Just as there's a type for objects, functions have a type as well. You can have function variables and call assigned functions through those variables. Mostly the function type is used in conjunction with other functions that use them as parameters.

You declare a function type just as any variable:

```
<data type> <variable name>, <another variable>, ..., <last variable>;
e.g.
    function my_func, *func_array;
```

Assigning actual function references to them, however, is a bit tricker. You can assign any kind of function to a function variable; efun, sfun or lfun is just the same. You can even assign external function references.

Assigning a function reference requires that the function already is defined, either itself or by a function prototype in the header. Let's assume for now that you're only interested in the simple reference to the function.

```
<function variable> = <function name>;
<function variable> = &<function name>();
e.g.
    my_func = allocate;
    my_func = &allocate();
```

Usage of the new function reference is done *just* as with the ordinary function call.

```
int *i_arr;

i_arr = allocate(5);  // Is the same as...
i_arr = my_func(5);   // ... using the function assignment above.
```

This will be enough for now. Later I'll explain how to create partial function applications, internal and external function declarations and how to use them in complex function combinations.

switch/case part 2

The LPC switch statement is very intelligent, it can also use ranges in integers:

```
public void
wheel_of_fortune()
{
    int i;

    i = random(10);     // Get a random number 0 - 9
                        // Strictly speaking, this local variable isn't
                        // necessary, it's just there to demonstrate the
                        // use and make things clearer. I could have
                        // switched on 'random(10)' directly instead if
                        // I had wanted to.

    switch (i)
    {
    case 0..4:
        write("Try again, sucker!\n");
        break;

    case 5..6:
        write("Congrats, third prize!\n");
        break;

    case 7..8:
        write("Yes! Second prize!\n");
        break;

    case 9:
        write("WOOOOPS! You did it!\n");
        break;

    default:
        write("Someone has tinkered with the wheel... Call 911!\n");
        break;
    }
}
```

catch/throw: Error handling during runtime

It happens now and then that you need to make function calls you know *might* result in a runtime error. For example you might try to clone an object (described later) or read a file. If the files aren't there or your privileges are wrong you will get a runtime error and execution will stop. In these circumstances it is desireable to be able to intercept the error and either display some kind of message or perform other actions instead. The special LPC function operator `catch()` will do this for you. It returns 1 (true) if an error occurs

during evaluation of the given function and 0 (false) otherwise.

```
int catch(function)
e.g.
    if (catch(tail("/d/Relic/fatty/hidden_donut_map")))
    {
        write("Sorry, not possible to read that file.\n");
        return;
    }
```

It's also possible to cause error interrupts. This is particularly useful when you want to notify the user of an unplanned for event that occured during execution. Typically you want to do this in the 'default' case of a switch statement, unless (naturally) you use default as a sort of catch−it−all position. In any case `throw()` will generate a runtime error with the message you specify. A `catch()` statement issued prior to calling the function that uses `throw()` will naturally intercept the error as usual.

```
throw(mixed info)
e.g.
    if (test < 5)
        throw("The variable 'test' is less than 5\n");
```

Array & Mapping references

In comp sci terms, arrays and mappings are used as ***reference by pointer*** and the other types as ***reference by value***. This means that arrays and mappings, unlike other variables, aren't copied every time they are moved around. Instead, what is moved is a reference to the original array or mapping. What does this mean then?

Well... simply this:

```
object *arr, *copy_arr;

arr = ({ 1, 2, 3, 4 });    // An array

copy_arr = arr;              // Assume (wrongly) that a copy_arr becomes
                             // a copy of arr.

// Change the first value (1) into 5.
copy_arr[0] = 5;
```

... Now... this far down the code it's logical to assume that the first value of `copy_arr` is 5 while the first value or `arr` is 1. That's not so however, because what got copied into `copy_arr` was not the array itself, but a reference to the same array as `arr`. This means that your operation later where you changed an element, changed that element in the original array which both variables refer to. `copy_arr` and `arr` will both seem to have changed, while in fact it was only the original array that both refered to that changed.

Exactly the same thing will happen if you use mappings since they work the same way in this respect.

So... how do you get around this then? I mean... most times you really want to work on a copy and not the original array or mapping. The solution is very simple actually. You just make sure that the copy is created from another array or mapping instead.

```
              _ This is just an empty array
             /
copy_arr = ({ }) + arr;
```

```
\_ This is the one we want to make unique
```

In this example `copy_arr` becomes the sum of the empty array and the `arr` array created as an entirely new array. This leaves the original unchanged, just as we wanted. You can do exactly the same thing with mappings. It doesn't matter if you add the empty array or mapping first or last, just as long as you do it.

LPC/Mudlib interafce

There's a lot of stuff you want to do, like handling strings and saving data to files, that's not exactly LPC. It's part of the 'standard function package' that most programming languages sport. This chapter will teach you the basics of how to do all the things you need in order to create LPC objects.

Objects in the game share a certain set of common properties, the ones you always can rely on to be there for any kind of object are these:

@bullet{creator}
: The object is created by someone. The identity of this creator is set depending on the file−system location of the source code. If the object resides in the directory of a domain−active wizard, the creator is said to be the name of that wizard. Otherwise the domain name is used. For mudlib objects the creator usually is `root` for admin objects and `backbone` for the rest.

@bullet{uid/euid}
: The ***uid*** (User ID) of an object defines the highest possible privilege level of an object. The uid itself is is only used to affect the ***euid*** (Effective User ID) of the same or another object. The euid is later checked in situations where the privilege of the object needs to be examined i.e. file access (reading/writing/removing) and object creation.

@bullet{living}
: In order for an object to be able to receive command−lists or issue commands it has to be ***living***.

Definition of standard and library objects

As I have explained earlier the gamedriver really knows very little about the actual game, actually as little as possible. Instead the mudlib is entrusted to take care of all that. So, what we have done is to try to work out what basic functionality is needed, things like how objects should interact with players, moving, light−level

descriptios etc. Then we have created basic object classes that implement these functionalities in actual code.

A domain wizard doesn't have spend endless hours trying to figure out how to make an object work in relation to others in respect to basic functionality. Instead he just makes his object inherit the standard object suitable for the task he wants to code. Then he just adds the bits and pieces of code to the object that is necessary to make it unique and do the things that are special for that particular object.

A consequence of this naturally is that all objects in the game rely 100% on the fact that a certain type of object (room, monster, gadget) has a certain set of common functionality. They simply *have* to have that in order to be able to interact in the agreed way, if they didn't, if people had different ways of solving the same problem, the objects would only work with a certain wizard's area and never outside of it. It would then not be possible use a sword all over the game, it wouldn't even be possible to move it around from place to place. Naturally this means that we enforce this unity, and therefore it is *impossible* to create (and use) objects that don't inherit these special objects. Sure, as you can see for yourself later, it is possible to create a sword that doesn't make use of the standard weapon object, but it is perfectly impossible to wield it...

The standard objects provide certain basic functionality that must exist in all objects and they also make a bit of sanity checking on values of certain variables, but the latter really is a very minor functionality.

There are standard objects for a lot of purposes, the most important one is `/std/object.c' though.

- /std/object.c: The base object class
- Standard object classes: A list of all object classes
- Library objects: Standard library objects

The base object class, /std/object.c

This is the all purpose object class. ALL objects in the game must inherit this object somewhere along the line if they are to be 'physically' present somewhere. Using any kind of standard object usually insures that this one is inherited as well, since they already make use of it.

The standard object defines the following conventions:

@bullet{inventory}
: An object can ***contain*** other objects. In reality that is nothing but a list of objects that are said to be held inside the object owning the list. However, it is very easy to visualize this as the inside of a bag, inside a room, inside a box etc.

@bullet{environment}
: The object that ***surrounds*** the object that is being used as reference. In other words the reference object exists in the inventory of the environment object. An object can have a multitude of objects in its own inventory, but it can only have one environment object. All objects start out with no environment.

@bullet{command set}
: A list of catch−phrases linked to functions that the object makes available to other so called ***living*** objects in the game either in the environment or inventory of itself. These living objects can issue such a catch−phrase and the command−giving object will execute the linked function.

@bullet{properties}
: Properties are a pure mudlib convenience. They really is nothing but a mapping with certain reserved names indexed to object variables that affect certain generally accessable states. Typical properties are weight, value and light−level, but also more abstract concepts like the ability to be dropped, taken or sold. The applicable set of properties vary from object type to object type. Wizards may add their own

properties if they wish, but they must then be careful to define names that won't mistakenly be used by other wizards for other purposes, or advertise the names so that people won't use them by mistake.

@bullet{light}
: An object has a certain light level. Usually it's just as any kind of object − not affecting the environment at all, but it's possible to have both light− and darkness−sources.

@bullet{weight/volume}
: These values determine how much an object weight and how much room they take. For 'hollow' objects like bags it also determines how much they can hold.

@bullet{visibilty}
: Some objects may be easier to find than others.

@bullet{names and descriptions}
: What the object is called and how a player will see it in the game.

Standard object classes

There exists a number of standard object classes for use in various situations. You will need to read the separate documentation on each and every one of them in order to fully learn to use them. However, this summary of the available classes will at least point you in the correct direction as you go.

As stated earlier you have to inherit most of these objects in order for your derived objects to work at all. However, it is possible to create new versions of some of them on your own. Again, I must emphasize that this is ***not*** a good idea since you then deviate from the common base of object functionality that is in use in the game. Problems that occur because of this actually are tricky to catch and track down since it isn't the first thing you suspect that someone has done.

`/std/armour.c'
: Armour of any kind

`/std/board.c'
: Bulletin boards

`/std/book.c'
: A book with pages you can open, turn and read

`/std/coins.c'
: The base of all kinds of money

`/std/container.c'
: Any object that can contain another

`/std/corpse.c'
: Corpse of dead monsters/players/npcs

`/std/creature.c'
: Simple living creatures, basically a mobile that can fight

`/std/domain_link.c'
: Use this as a base to preload things in domains

`/std/door.c'
: A door that connects two rooms

`/std/drink.c'
: Any type of drink

`/std/food.c'
: Any type of food

`/std/guild (directory)'
: Guild related objects (the guild and the shadows)

`/std/heap.c'
: Any kind of object that can be put in heaps

`/std/herb.c'
: Herbs

`/std/key.c'
: Keys for doors

`/std/leftover.c'
: Remains from decayed corpses

`/std/living.c'
: Living objects

`/std/mobile.c'
: Mobile living objects

`/std/monster.c'
: Monsters of any kind

`/std/npc.c'
: A creature which can use 'tools', i.e. weapons.

`/std/object.c'
: The base object class

`/std/poison_effect.c'
: Handle effects in poison of any kind

`/std/potion.c'
: Potions

`/std/receptacle.c'
: Any kind of closable/lockable container

`/std/resistance.c'
: Handle resistance against various kinds of things

`/std/room.c'
: Any kind of room

`/std/rope.c'
: Rope objects

`/std/scroll.c'
: Scrolls

`/std/shadow.c'
: Used as base when creating shadows

`/std/spells.c'
: Spell objects, tomes etc

`/std/torch.c'
: Torches/lamps etc

`/std/weapon.c'
: Any kind of weapons

Standard library objects

These objects are more of the order of 'help' clases. They don't qualify as objects in their own right, but they provide neat functionality for what they do.

`/lib/area_handler.c'
: Big general areas

`/lib/bank.c'
: Provides money changing support

`/lib/cache.c'
: Cache for frequent file access

`/lib/guild_support.c'

Support functions for guilds
`/lib/herb_support.c'
Support functions for herbs
`/lib/more.c'
More (file browsing) functionality
`/lib/pub.c'
Bar/pub functionality
`/lib/shop.c'
Shops of all kinds
`/lib/skill_raise.c'
Training of skills
`/lib/store_support.c'
Support functions for stores
`/lib/time.c'
Time handling routines
`/lib/trade.c'
Trade related support functionality

How to obtain object references

Objects, as previously described, comes in two kinds − ***master objects*** and ***clones***. In general you tend to use cloned objects. At least for objects that are being 'handled' in the game, objects that you can move about, touch, examine etc, or any object that exist in more than one copy. Making exclusive use of only the master object is usually only done for rooms, souls or ***dameon*** objects of various kinds.

Naturally *any* object in the game must have a master object. An object is loaded into memory and the master object created when a function (any function call, even to a non−existing function) is called in it. Cloning it just makes identical copies of it. If you destroy the master object, the gamedriver will have to load it again later before making any new clones. Naturally this is what you do every time you have made changes to the object that you want to become active. Destroying the master object won't change the already existing clones, of course. You'll have to replace them separately.

The mudlib in fact works so that ***loading*** an object is made by calling a non−existing function in the object and ***updating*** it simply destroys the master object.

How to get the object references then? Well, that depends on the situation. An object reference is either an object pointer or a string path, referring to the object source in the mud filesystem. Obtaining them is different depending on the situation however. Let's go through them all.

- Relative references: References relative to the current object
- Creating objects: How to create an object
- Finding relative refs: Find a reference relative to another object
- Interactive object refs: References to interactive objects
- Destroying objects: How to get rid of objects

Object references relative to the current object

[this_object, previous_object, calling_object]

An object can always get the object reference to itself. Use the efun `this_object()`:

```
object this_object()
e.g.
    object ob;

    ob = this_object();
```

In order to find out which object called the currently running function in an object using an external call, you can use the efun 'previous_object()':

```
object previous_object(void|int step)
e.g.
    object p_ob, pp_ob;

    p_ob = previous_object();     // The object calling this function.
    pp_ob = previous_object(-2);  // The object calling the object
                                  // calling this function.
```

If you supply no argument or −1, the function will return the immediately previous object that called. Decrementing the argument further will return even more previous callers, i.e `previous_object(-4)` returns the object that called the object that called the object that called your object. If indeed the chain of calling objects was that long. When you exceed the length of the calling chain beyond the first object that made a call, the function will return 0.

As I hope you noticed, this call only checks for external calls, not internal. There is a corresponding efun that works just the same but for any type of call (internal or external) that has been made:

```
object calling_object(void|int step)
```

The usage is the same however.

So... how do you know if the object reference you just received is a valid object or not (i.e. 0 or something else)? Well, use the nice efun `objectp()` as described earlier. It returns 1 if the argument is a valid object pointer and 0 otherwise.

```
int objectp(mixed ob)
e.g.
    if (objectp(calling_object(-2)))
        write("Yes, an ob calling an ob calling this object exists!\n");
    else
        write("No such luck.\n");
```

Creating objects

[setuid, getuid, seteuid, geteuid, creator, set_auth, query_auth, clone_object]

First of all you must make sure that the object that tries to create a new object has the privileges required to do so. The rules are pretty simple actually: An object with a valid euid can clone any other object. A valid euid is anything except 0. The euid 0 is the default uid and euid on creation of an object, and it's used as meaning 'no privileges at all'.

However, usually the choice of euids you can set is pretty limited. If you're a wiz it's usually limited to your own name. A Lord can set the euid in an object to be his, or any of the wizard's in the domain (unless one of the wizards is an Archwiz, then that one is excempt as well). And naturally objects with 'root' uid can set any euid they like.

So... the uid of the object determines what choice of euids you have. You set the uid to the default value by adding this sfun call:

```
void setuid()
e.g.
    setuid();
```

Simple eh? Doing that sets the uid to the value determined by the location of the object source−file in the mud filesystem. The rules for this is the same as for the ***creator*** value described earlier. You can get the creator value of an object with the sfun `creator()`, it simply returns the string `setuid()` would use for that object.

```
string creator(mixed reference)
e.g.
    string my_creator;

    my_creator = creator(this_object());
```

To get the actual uid value that is currently used, you the sfun `getuid()`

```
string getuid()
e.g.
    string curr_uid;

    curr_uid = getuid();
```

So.. the uid is now set to the highest privilege giver. The euid however, is still 0. Since the euid determines the actual privileges used in an object this means that the object still has *no* privileges at all.

To set the euid you use the sfun `seteuid()`, the argument given will be set as euid if allowed (it's tested). The function returns 0 on failure and 1 on success. If you don't send any argument, the euid is set to 0, 'turning it off' so to speak.

```
int seteuid(void|string priv_giver)
e.g.
    if (seteuid("mrpr"))
        write("Yes! I'm the ruler of the UNIVERSE!\n");
    else
        write("Awwwww....\n");
```

Naturally there's a corresponding sfun to return the current euid:

```
string geteuid()
e.g.
    write("The current euid = " + geteuid() + "\n");
```

The sfuns `setuid()`, `getuid()`, `seteuid()` and `geteuid()` are all using the efuns `set_auth()` and `get_auth()`. They are used to manipulate a special authority variable inside the object in the gamedriver. The gamedriver will call a validizing function in the master object (security) if you try to use `set_auth()` to make sure that you are privileged to do so. The reason is that it's possible to store any kind of string in the authority variable, and the way we use it is merely a convention, something that we have decided is the best way of solving security.

When you try to perform a privileged operation, like writing to a file or cloning an object the gamedriver calls other special functions in the master object to make sure you have the right privileges. They all depend on that

the information stored in the authority variable is formatted in the special way we want for it to work properly. Due to this fact you are not allowed to use `set_auth()` in any other way than already is allowed by `setuid()` and `seteuid()`, so there's really no use in doing that at all. `query_auth()` is not protected but you won't find much use for that information anyway.

The information stored in the authority variable is simply the uid and euid separated by a colon.

Now that we know how to give privileges to an object, let's find out how to make it ***clone*** others! The efun used is called `clone_object()`, it loads and creates an object from a source file. If the cloning should fail, due to programming mistakes for example, an error message will be given and execution of the current object aborted.

```
object clone_object(string obref)
e.g.
    object magic_ring;

    // Set the object privileges so that it's possible to clone
    setuid();
    seteuid(getuid());

    // Actually clone the object
    magic_ring = clone_object("/d/Domain/wiz/magic_ring");
```

Naturally you only have to set the uid/euid of an object *ONCE* in an object and not every time you want to perform a privileged operation. The most common procedure is to put thes uid/euid setting calls in a function that is called when the object is first created, but more about that later.

Now... when arrays or mappings were created they existed as long as any variable used them. If the variable was set to 0, the data they contained was scrapped as well. Is this true for objects as well? ***NO!*** It's not. The object will remain in the game as long as the gamedriver is running, unless you explicitly destroy it.

Finding references relative to another object

[file_name, find_object, object_clones, find_living, set_living_name, MASTER_OB, IS_CLONE]

As stated object references either are strings or object pointers. Turning an object reference to a string is done with the efun `file_name()`:

```
string file_name(object ob)
e.g.
    write("This object is: " + file_name(this_object()) + "\n");
```

The string that pops out of `file_name` is the string representation of the object reference pointer. It's given as `<file path>#<object number>`, for example `"/d/Domain/wiz/magic_potion#2321"`. This string is a valid object reference to that specific object as well.

To turn a string object reference into an object pointer reference you use the efun `find_object()`.

```
object find_object(string obref)
e.g.
    object the_ob;

    // The master object
    the_ob = find_object("/d/Domain/wiz/magic_potion");
```

```
    // The specific clone
    the_ob = find_object("/d/Domain/wiz/magic_potion#2321");
```

If the function doesn't find the object (the path might be wrong, the specified clone might not exist or the object might not be loaded), it returns 0.

Sometimes it's useful to find all the clones of a specific object. The efun for that is `object_clones()`. It will return an array holding all clones of the master object the object reference indictes. This means that you can give either a master object or an object clone pointer as argument. However, be a bit careful here. If the object was updated and your provide the master object as argument, you will get a list of all the 'new' clones. If you give an old object as argument you will get a list of all contemporary objects, the objects of that 'generation'. If no clones can be found, an empty array is returned.

```
object *object_clones(object obref)
e.g.
    object *ob_list;

    ob_list = object_clones(find_object("/d/Domain/wiz/magic_potion"));
```

Some objects are ***living***. In the game this denotes the fact that the objects can be attacked and (perhaps) killed and that they want to receive command updates from objects that turn up either in the environment or the inventory of the object itself. Living objects have the option of registering themselves in a special list in the gamedriver. This is done in order to make them easier to find. The special efun `find_living()` looks for a named living object in the internal list of names.

```
object *find_living(string name, void|int 1)
e.g.
    object balrog_ob, *bals;

    // Search for the 'balrog' monster in the game.
    balrog_ob = find_living("balrog");
```

If you give '1' as second argument, the efun will return a list of all objects with that name found instead.

```
bals = find_living("balrog", 1);
```

If no living object with the given name can be found, 0 is returned.

In order for the name to become part of the list of names, the object itself must add the name to the central list with the efun `set_living_name()`.

```
void set_living_name(string name)
e.g.
    // This is part of the 'create()' function of the balrog above.
    set_living_name("balrog");
```

Remember that if you have several objects with the same name, `find_living()` in the single object mode will randomly return one of them.

For your own sake you ought to reserve the use of npc names with the special 'banish' command in the game, so that no players turn up with the same name as you npc. If that happens things are very likely to get confused...

In order to get the master object reference of an object you have a pointer to, you can convert it to a string, then strip off the object specifying bits. However, there's already a macro doing that in the standard package `/sys/macros.h'. Simply add the line #include <macros.h> to the top of your file and use the macro MASTER_OB.

```
string MASTER_OB(object ob)
e.g.
    string master;

    // Assume that /sys/macros.h is included in this file.
    master = MASTER_OB(find_living("balrog"));
```

As stated, this returns the string reference to the master object, if you particularly need the object reference just get it with find_object() given the just established object path as argument.

A clone is easiest distinguished from the master object by comparing the object reference strings. The macro IS_CLONE does that for you, also available in `/sys/macros.h'. The macro works on this_object() and takes no argument

```
int IS_CLONE
e.g.
    if (IS_CLONE)
        write("I am a clone!\n");
```

Object references to interactive objects

[find_player, this_interactive, this_player]

If you are looking for a particular player, you could look for him with find_living() and then just make sure it's an interactive object. However, it's a lot quicker to use the efun find_player() that works just the same with the exception that there can only be one player with a given name; if he's in the game you will get the object reference to him and no other.

```
object *find_player(string name)
e.g.
    object fat_one;

    fat_one = find_player("fatty");

    if (objectp(fat_one))
        fat_one->catch_msg("Hail thee, bloated one!\n");
    else
        write("Nope, no such luck.\n");
```

Very often you want to know which player issued the command that led to the execution of a specific function. The efun this_interactive() will return the object reference to that player. If the execution chain was started by an independent non−player object, 0 is returned.

However, more often you're not interested in who actually started the chain, but rather who the object is supposed to direct its attention at. That object is returned by the efun this_player(). In other words, while the object might be expected to turn its attentions (command lists, output messages, object effects etc) to the object given by this_player(), it might be another player given by this_interactive() that actually started the execution chain in the object. The value of this_interactive() can never be manipulated by objects in the game, this_player() on the other hand can be set at will. More about that

later.

```
object this_player();
object this_interactive();
e.g.
    object tp, ti;

    tp = this_player();
    ti = this_interactive();

    if (objectp(ti))
    {
        if (ti != tp)
        {
            tp->catch_msg("Zapppp!\n");
            ti->catch_msg("Zapped him!\n");
        }
        else
            ti->catch_msg("Fzzzzz...\n");
    }
```

Destroying objects

[destruct, remove_object]

Sooner or later you will want to get rid of an object. The efun you use is `destruct()`. However, the gamedriver *only* allows the object that issued the `destruct()` efun to actually be removed. This means that every object need to have a function that you can call from another object in order to be able to destroy it from without. If the object doesn't contain the `destruct()` efun, it will remain for the duration of the game.

Well, actually there exists a backdoor that allows you to destroy any object, but it's a command you have to issue manually. You can't use it in a program.

However, the standard object base − which is being discussed in more detail later − does define a function called `remove_object()` that you can call to destroy the object. Since all objects actually in the game ***MUST*** inherit the standard object you can rely on having that function there. It's possible to mask it, thereby blocking that function. However, masking `remove_object()` is tantamount to sabotage so please don't even think about it. The reason the function is maskable is so that you should be able to add last−second cleanup code, not so that you should be able to render the object indestructable.

```
void remove_object()
e.g.
    void
    remove_balrog(string bal_name)
    {
        object bal;

        bal = find_living(bal_name);
        if (objectp(bal))
            bal->remove_object();
    }
```

If you use the `destruct()` efun directly or call `remove_object()` in the object itself, make ***DOUBLE*** sure that no code is being executed afterwards. You see, execution isn't aborted on completion of the destruction, the object is just ear−marked as ***destructed***, actual removal is done when execution of it is finished. This means that function calls or commands issued after destruction might give rise to runtime errors

in other objects.

```
void destruct()
e.g.
    void
    destruct_me()
    {
        write("Goodbye, cruel world!\n");
        destruct();
    }
```

When an object is destructed, *ALL* object pointers (not string references) in the game pointing at the destructed object are set to 0. Due to this fact it's usually sensible to make sure that an old object reference still is valid before doing anything with it. You never know, it just might have been removed since you obtained it.

if an object contains other objects in its inventory when it is destructed, those objects will be destructed with it. The exception is interactive objects, players. If you update a room you are effectively destructing it, if it has players in it they will be moved to their respective starting locations. If the room *is* the start location or if there is a problem with moving them (buggy start location or impossible to move them) those players will be destructed as well. In any circumstances you should always be able to rely on an object being destructed when ordered to.

The one time there's problems with this is when the function `remove_object()` has been overridden and has a bug in it. That might just abort the process and cause problems.

Object−inherent command handling

[init, add_action, enable_commands, disable_commands, living, command, commands, get_localcmd, query_verb, notify_fail, update_actions]

By now you know that nothing ever is really simple in this game. To confuse the issue when dealing with commands we actually have two different types of commands. One kind is the one that we will talk about here, commands that are defined by objects that you can touch or examine. The other kind is so called 'soul' commands. The soul commands is a pure mudlib convenience though. They are described later in chapter three.

Object−added commands work like this: Upon entering an object, such as a room, a special lfun called `init()` is called in the room and in all other objects in the inventory of the room. The `init()` function is actually an ordinary function that you could use for any purpose, but the *intended* purpose is to have the command−adding efun `add_action()` there. In other words, when you enter an object of any kind, the commands of that object as well as those of the other object in the same inventory are added to your set of commands.

The `add_action` command ties an action word to a function in the object. Upon typing the special word as the first word on the line, the function gets called with any other words you might have typed as arguments. The third argument can be given as '1' if you want the gamedriver to trigger on just a part of the word. For example if you have the action word 'examine' and want to allow 'exa' or 'ex' as well.

```
add_action(function func, string action, void|int 1)
e.g.
    init()
    {
```

```
        /*
         * The functions 'do_bow()' and 'leave_game()' are defined
         * somewhere in this object. However, if it's done later than
         * this function, they must exist as prototypes in the header.
         */
        add_action(do_bow, "bow");          // Better get used to seeing
        add_action(&leave_game(), "quit");  // different kinds of function
                                            // reference declarations.
    }
```

Is this true for any kind of object then? Will any object receive this set of commands? No. Just ***living*** objects. An object is made living by the efun `enable_commands()` and dead, or inert, with the efun `disable_commands()` Note carefully that ***living*** in the gamedriver only means ***being able to receive and execute commands***, in the mudlib it means a bit more.

Use these efuns whenever you want to switch on or off the command handling capabilities. However, remember that if the object already is moved into a room when you turn on living status, it will have no command lists. You will have to move it out/in to the room again in order for it to pick up all actions.

You can check if an object is living or not with the efun `living()`.

```
enable_commands()
disable_commands()
int living(object ob)
e.g.
    public void
    toggle_living_status()
    {
        if (living(this_object()))
            disable_commands();
        else
            enable_commands();
    }
```

Actions can only be added and maintained by objects that exist in the inventory or environment of an object. If the object is moved from the action−defining object's presence, the action is removed as well.

As you now understand, the gamedriver expects the function `init()` to be defined in any object that wishes to add actions to living objects. Please be careful how you use the function though. If you for example use the `init()` function to test if an object belongs there or not, and then move it out if it doesn't, you'll likely get into trouble. The reason is that if you add actions after having moved it, you will in fact be adding actions to a non−present object. The gamedriver will notice this and you will have an error. I would like to advice you ***not*** to use the `init()` function for any other purpose than adding actions. It's allowed to test the object that calls the `init()` function to determine if you should add actions to it or not (if you limit access to some actions), but avoid any other kind of code.

I'm sorry to say that the mudlib itself doesn't always conform to this rule, there's objects here and there that cheat. However, there's no reason why you should code things badly just because we did. :)

In most objects that inherit standard base objects it's necessary to call the parent `init()` as well, since otherwise you'll override it and thereby miss lots of important actions. Just put the statement `::init();` first in your init, before your `add_action()` statements, and all will be well.

To execute a command in a living object you use the efun `command()`.

```
int command(string cmd)
e.g.
    command("sneeze");
    command("wield sword in left hand");
```

If you're doing this for mortals (which most often is the case) you're wise to use this construction instead. The reason is that the dollar sign will evade the internal alias mechanism so that it isn't fooled by an unfortunate macro.

```
int command(string cmd)
e.g.
    command("$sneeze");
    command("$wield sword in left hand");
```

The commands naturally only work if they have been added to the living object by other objects in the environment or inventory. To get a list of the available commands you can use the efuns `commands()` or `get_localcmd()` depending on what kind of information you want. `commands()` return an array of arrays with all the commands of a specified object, containing not only the command word, but also what object defines it and which function is called. `get_localcmd()` is simpler, returning only an array with the command words. If no object is specified, `this_object()` is used by default. See the manual entry for `commands()` to get the specification of what the command array contains.

```
mixed commands(void|object ob)
string *get_localcmd(void|object ob)
```

If you use one function for several command words it becomes necessary to find out exactly what command word was used. You use the efun `query_verb()` for that.

```
string query_verb()
e.g.
    init()
    {
        ::init();
        add_action(&my_func(), "apa");
        add_action(my_func, "bepa");
        add_action(&my_func(), "cepa");
    }

    public int
    my_func()
    {
        switch (query_verb())
        {
        case "apa":
            < code >
            break;

        case "bepa":
            < code >
            break;

        case "cepa":
            < code >
            break;
        }
        return 1;
    }
```

Action functions should return 1 if they were properly evaulated, i.e. if the function called was the right one with the right arguments. If you return 0, the gamedriver will look for other actions with the same command word and try those, until one of them finally returns 1, or there's no more to test. There's a special efun called `notify_fail()` that you can use for storing error messages in case no function 'takes' the command. Instead of giving the very useless text 'What ?' when the player types the command you can give him some better info. If there are several action commands using the same command word who all fail, the one who *last* called `notify_fail()` will define the message actually used.

```
notify_fail(string message)
e.g.
    public void
    init()
    {
        ::init();
        add_action(&do_bow(), "bow");
    }

    public int
    do_bow(string who)
    {
        if (!present(find_player(who)), environment(this_player()))
        {
            notify_fail("There's no " + who + " here to bow to!\n");
            return 0;
        }

        < bow code >

        return 1;
    }
```

If you are absolutely certain that the command given was directed only to your object and you want to stop execution there, even if your object finds an error with the command (arguments or context or whatever), you can return 1. However, then you must use another method to display error messages. The text stored with `notify_fail()` is *only* used if you return 0.

If your object changes the available set of actions during execution and you want the surrounding living objects to update their command set, you call the efun `update_actions()` for the object in question. If you don't specify any object `this_object()` is used by default. What happens is that all surrounding objects discard their command sets from the specified object and call `init()` in it again to get the new set.

```
update_actions(object ob)
```

Alarms: Asynchronous function execution

[set_alarm, remove_alarm, get_alarm, get_all_alarms]

Somtimes it's desirable to postpone execution of code a while and sometimes you want things to happen regularly. The gamedriver counts something called ***evaluation cost***, or ***eval cost***. It's simply a way of measuring the amount of CPU cost an object uses. Any given object is only allowed a certain amount of eval cost per execution chain. When that amount is used up, the object aborts. How the eval cost is computed isn't very important, it's set so that the game shouldn't be held up too long. However, the existance of eval cost makes a bit of special programming necessary. When you have very heavy computations you need to do, they simply won't fit withing the maximum allowed eval cost, so you need to cut the job up in chunks and do it bit by bit.

All of this adds up to a need for a function that allows you to do things regularly, or with delays. The alarm functionality will do this for you. The efun `set_alarm()` will allow you to create a delayed alarm, possibly repeating, that will call a given function as you decide.

```
int set_alarm(float delay, float repeat, function alarm_func)
remove_alarm(int alarm_id)
mixed get_alarm(int alarm_id)
mixed get_all_alarms()
```

The function returns a unique alarm number for that alarm and that object that you can use later to manipulate the specific alarm. You can retrieve info for the alarm with the efun `get_alarm()`, remove it with `remove_alarm()` or even get info about all alarms in an object with the efun `get_all_alarms()`. The latter function is mostly used when you either haven't bothered to save the alarm ids, or when you want to display info about the object. The efun `set_alarm()` allows you both to define a delay until the function is called the first time, and a delay between repetitive calls. Every alarm call will start with an eval cost at 0.

A small word of caution here... Since the function gets called asynchronously in respect to a user of the object, both `this_player()` and `this_interactive()` might return undefined values. Sometimes 0, sometimes the object you expect, sometimes another value. So, don't rely on what they return, instead stick the object you want to use in a variable *before* starting the sequence and uese that. Remember this since some efuns rely on a defined `this_player()` value.

IMPORTANT! READ THIS CAREFULLY!

It's very easy to fall to the temptation to split a heavy job into several alarm calls with fast repetition rates. However, this is ***NOT*** the intended use for this efun. A deadly sin is to have an alarm function that generates repeating alarms within a repeating alarm. The amount of alarms will then grow exponentially and the ***ENTIRE GAME*** will stop almost immediately. This is so incredibly stupid as to be a demoting offense, so make sure you do things ***RIGHT*** the first time. In general, delays between repeating alarms should be greater than one second, preferrably two, as well as delays to single alarms.

The alarm functions will be demonstrated more extensively in chapter three.

The inventory and the environment

[move_object, move, enter_inv, enter_env, leave_inv, leave_env, environment, all_inventory, deep_inventory, id, present]

As described earlier, an object defines an ***inside*** as well as an ***outside***. The outside, or ***environment*** can only be one object, while the inside, or ***inventory***, can contain many objects.

A newly cloned object ends up in a sort of limbo, without an environment. In order for an object to actually enter the simulated physical world of the game it has to be moved there. However, not all objects can be moved around. In order for the game to work ***ANY*** object that wants to be inserted somewhere or have objects inserted into it ***MUST*** inherit `` `/std/object.c' `` somewhere along the way in the inheritance chain. Why this limitation? Well, the reason is that the standard object defines a number of handy functions we rely on all objects in the game to define.

The most important of these lfuns are:

`move()'
: Move an object to another object and handle weight/volume accounting. Returns success code. This function is responsible for calling the following:

`enter_inv()'
: This function is called in an object when another object moves inside it.

`leave_inv()'
: This function is called in an object when another object moves out from it.

`enter_env()'
: This function is called in an object upon entering the environment of another object.

`leave_env()'
: This function is called in an object upon leaving the environment of another object.

NB! The above ***ONLY*** works if you use the lfun `move()` in the object to move them around. That's why it's so important that you do it this way and not by the efun that actually performs the move.

The efun used in the `move()` lfun is `move_object()`. ***BUT***, remember when doing that none of the object internals like light, weight or volume is updated. As previously stated the efun fails if the object you want to move, or move to, doesn't inherit `` `/std/object.c' ``. Furthermore the efun can only be used from within the object that wants to move, it can't be used to move another object. The same goes for the `move()` lfun, naturally.

In order to get the enclosing object reference you use the efun `environment()`. As I have said before all objects have no environment on creation, it's only after they have been moved somewhere that it gets a proper environment. Once an object has been moved into another object it can't be moved out into limbo again, i.e. you can't move it to '0'. The objects in the game you can expect not to have an environment are either rooms, souls, shadows or daemon objects of one kind or another.

You have two efuns to chose between when it comes to finding what's in the inventory of an object. The efun `all_inventory()` returns an array with all the objects in the inventory of a specified object, while the efun `deep_inventory()` return an array with all objects recursively found in the inventory, i.e. not only the objects you'll find immediately but also the objects *in* the objects in the inventory, and so on.

```
object *all_inventory(object ob)
object *deep_inventory(object ob)
e.g.
/*
 * This function dumps the inventory of Fatty on the screen,
 * either just what's immediately visible or all depending
 * on a given flag.
 */
void
fatty_say_aaah(int all)
{
    object fatty_ob, *oblist;

    if (!objectp((fatty_ob = find_player("fatty"))))
    {
        write("Sorry, Fatty isn't in the game today.\n");
        return 0;
    }

    oblist = all ? deep_inventory(fatty_ob) : all_inventory(fatty_ob);

    write("The " + (all ? "entire " : "") +
                " content of Fatty's bloated tummy:\n");
```

```
    dump_array(oblist);
}
```

So... how do you go about to determine if a specific object actually is present in the inventory of another object? Well, the base object `/std/object.c` define both names and descriptions in objects, as described before. It also defines a special function called `id()` that, given a name, checks all given names to an object for a match and returns 1 if the object has that name. The efun `present()` takes a name or object reference and searches one or more object's inventories for the presence of the named or specified object. If you specify the object to search for as a name string it will use the previously mentioned `id()` function to determine if the object is the right one or not for all objects it examines. The execution of the function is aborted as soon as it finds one that fits the description. That means that if there are several objects fitting the search pattern you will only get one of them.

```
object present(object ob|string obref, object *oblist|object ob|void)
e.g.
/*
 * Look for donuts in Fatty
 */
void
find_donut()
{
    object fatty_ob;

    fatty_ob = find_player("fatty");

    // Can't find Fatty!
    if (!objectp(fatty_ob))
    {
        write("Fatty isn't in at the moment, please try later.\n");
        return;
    }

    if (present("donut", fatty_ob))
        write("Yes, Fatty looks happy with life at present");
    else
        write("If I were you, I'd keep out of Fatty's " +
                  "reach until he's fed.\n");
}
```

If you don't give any second argument to `present`, it will look for the specified object in the inventory of `this_object()`, i.e. the object itself. If the second argument is given as an array, the function will look for the specified object in all of the objects in the array. If no fitting object is found, 0 is returned.

String functions

[break_string, capitalize, lower_case, sprintf, strlen, wildmatch]

In a gaming environment based on text, it's natural to expect that we've gone into a bit of trouble in making string handling functions both easy to use and versatile. As you already know, strings can be added together using the + operator, even mixing in integers without any special considerations. Floats and object pointers have to be converted however, floats with the special `ftoa()` efun (described later) and object pointers with the `file_name()` efun that I described earlier.

One of the most interesting properties of strings, apart from what they contain, is the length. You find that with the efun `strlen()`. Since it accepts ints as well (returning 0 for them) you can use it to test

uninitialized string variables as well.

```
int strlen(string str)
e.g.
    string str = "Fatty is a bloated blimp";
    write("The length of the string '" + str +
                "' is " + strlen(str) + " characters.\n");
```

Often you will want to capitalize names and sentences for output to the screen. You do that with the efun `capitalize()`, it will only turn the first character in the string to upper case. The converse of this function is the efun `lower_case()`, however, it turns the *entire* string into lowercase and not only the first character.

```
string capitalize(string str)
string lower_case(string str)
e.g.
void
// Present a given name on the output, formatted properly
present_nice_name(string name)
{
    string new_name;

    // Assume name = "fAttY"
    new_name = lower_case(name);
    // Name is now = "fatty"
    new_name = captialize(name);

    write("The name is: " + name + "\n");

    /* The result is:

       The name is: Fatty
     */
}
```

Sometimes it's desirable to break up a string in smaller pieces, just to present a nicer output. The efun `break_string()` will do that for you. It can even pad spaces in front of the broken strings if you want that. What it does is simply to insert newlines after whole words where you have indicated you want to break it up. The third argument specifying either space pad length or a string to pad with, is optional.

```
string break_string(string str, int brlen, int indlen|string indstr|void)
e.g.
    string str = "This is the string I want to present in different ways.";

    write(break_string(str, 20) + "\n");
    write(break_string(str, 20, 5) + "\n");
    write(break_string(str, 20, "Fatty says: ") + "\n");

    /* The result is:

       This is the string I
       want to present in
       different ways.

            This is the string I
            want to present in
            different ways.

       Fatty says: This is the string I
       Fatty says: want to present in
```

```
      Fatty says: different ways.
    */
```

You will very often want to present information stored in variables. As shown you can do that by converting the contents to strings and then just print the strings. Integers don't even have to be converted, you just add them on with the +−operator. However, what you get then is something that's not very well formatted, you'll have to do that yourself. Particulary if you try to produce tables this is a nuisance, having to determine the length of strings and add on a certain amount of spaces depending on this length and so on. Instead of doing this you can use the efun `sprintf()`.

What `sprintf()` does is simply to take a ***format***−string that describes how you want the resulting string to look and put in the contents of the given variables according to your specifications. The result is a string that you then can present on the screen with `write()` for example.

All characters in the format string will be copied to the resulting string with exceptions of the special pattern ***%<width spec><type spec>***. The width specifier can contain a field width parameter, simply an integer that specifies the width of the ***box*** you want to put it in and if you want it left− or right−aligned. A positive number denotes right−alined insertion and negative number left−aligned. If you omit the width specifier the variable will be inserted in a box exactly the width of its contents. The type specifier is one or more characters defining what kind of variable you want to have inserted.

`d'
`i'
: The integer argument is printed in decimal.

```
string str;
int a;

a = 7;
str = sprintf("test: >%-3d%i<", 1, a);

write(str + "\n");

// The result is:
// test: >1  7<
```

`s'
: The argument is a string.

`c'
: The integer arg is to be printed as a character.

`o'
: The integer arg is printed in octal.

`x'
: The integer arg is printed in hex.

`X'
: The integer arg is printed in hex (in capitals).

`O'
: The argument is an LPC datatype. This is an excellent function for debug purposes since you can print ANY kind of variable using this specifier. e.g.

```
write(sprintf("1:%d 2:%s 3:%c 4:%o\n5:%x 6:%X 7:%O\n", 5,
              "hupp happ", 85, 584, 32434, 85852, strlen));

// The result is:
// 1:5 2:hupp happ 3:U 4:1110
// 5:7eb2 6:14F5C 7:<<FUNCTION &strlen()>>
```

This specifier is also the only one you can use for denoting floats right now.

Now, these were all the available type specifiers with a few width specifiers given as examples. However, there's a lot more of them.

`' ''
: The integer argument is padded with a space if it is positive. This way you can write neat tables with both positive and negative numbers without having to take special measure to handle the minus−sign.

`+'
: Positive integer arguments are padded with a plus sign.

`'X''
: The character(s) within the quotes is used to pad the argument to the field size.

`|'
: The argument will here be centered within the field size.

`#'
: This is table mode. The result will be a list of `\n`−separated words in a table within the field size. This naturally only is meaningful with strings.

```
write((sprintf(">%19|s<\n", "Fatty the blimp")));

// The result is:
// >  Fatty the blimp  <
```

`='
: This specifier is only valid for strings. It outputs the result in columns if the arguments are wider than their field size.

`*'
: Specifies to sprintf to use the corresponding argument as field size. If you combine this with table mode you will get neat tables formatted after the biggest argument.

`@'
: The argument is an array. You must naturally combine with with a type specifier denoting the specific type in the array.

Very often you want to find out if a certain substring is part of a greater string. You're not interested in exactly where the string is, just that it is there. For that purpose you want something that closly resembles the UNIX shell approach to string matching. The efun `wildmatch()` will do this for you. It simply return 1 if a specified substring is part of a specified main string, and 0 otherwise. The substring can contain the simple UNIX pattern matching symbols.

@bullet{}*
: matches any number of any characters

@bullet{?}
: Matches any single character

@bullet{[xyz]}
: Matches any characters in xyz

@bullet{[^xyz]}
: Matches any characters not in xyz

@bullet{\c}
: Matches c even if it is special

```
int wildmatch(string pattern, string str);
e.g.
```

```
    // Anything ending with .foo
    wildmatch("*.foo", "bar.foo") == 1
    // Anything starting with a, b or c, containing at least
    // one more character
    wildmatch("[abc]?*", "axy") == 1
    wildmatch("[abc]?*", "dxy") == 0
    wildmatch("[abc]?*", "a") == 0
```

Bit functions

[clear_bit, set_bit, test_bit]

Sometimes it's desireable to store lots of simple `on/off'−type information. The quick and dirty approach is then to allocate one integer for each of this information bearers and use them to hold either a one or a zero to indicate the state. This makes for easy access and easy understanding, but it's a pain when you want to store the info and it takes a lot of memory.

Instead you can use strings where every bit in a character (8 per char) can hold an information state of the on/off kind. The max number of bits right now in a string is something like 1200 = a string length of 150 characters. However, I doubt you'll ever need to store that many states.

You set the bits with the efun `set_bit()` which takes two arguments, the first is the string that actually contains the bits and the second is an integer specifying exactly what bit you want to set. Remember that the first bit is bit number 0. To clear a bit you use the efun `clear_bit()` that works analogous to `set_bit()`. When you need to test a bit you use the efun `test_bit()` which simply takes the same arguments as the other efun but returns 1 or 0 depending on whether the tested bit was set or not.

You don't have to allocate a string in advance when you use `set_bit()`. Both `set_bit()` and `clear_bit()` return the new modified string, and in case it's not wide enough it will be extended by `set_bit()`. However, `clear_bit()` will not shorten it automatically.

```
string set_bit(string bitstr, int the_bit)
string clear_bit(string bitstr, int the_bit)
int test_bit(string bitstr, int the_bit)
e.g.
    // Set bit 23 in a new bitfield.
    string bf;

    bf = "";
    bf = set_bit(bf, 22);

    // Clear bit 93 in the same bitfield
    bf = clear_bit(bf, 92);

    // Test bit 3
    if (test_bit(bf, 2))
        write("Set!\n");
    else
        write("Clear!\n");
```

Time functions

[time, ctime, file_time, last_reference_time, object_time, convtime]

All time measurements in UNIX, and hence in the mud, are measured starting at Jan 1, 1970 for some obscure reason. Perhaps the creators of this system figured that, from a computer point of view, there's no reason ever to need to store a timestamp of an earlier date. In any case that's how it is. Timestamps are integers and measure the time in seconds from that previously mentioned date.

The simple efun `time()` will return the current time. You can either use it as it is or convert it to a printable string with the `ctime()` efun. To find the creation time of a file, you use the efun `file_time()`, of an object the efun `object_time()`.

Sometimes it's desirable to know when the object last was referenced, i.e. when it last had a function called in it. If you then, as the first instruction in the function call `last_reference_time()` you will get that time. However, remember that it naturally is set to the current time as soon as that is done.

```
int time()
string ctime(int tm)
int file_time(string obref)
int object_time(object ob)
e.g.
    write("This object was last referenced at " +
        ctime(last_reference_time()) + "\n");
    write("The time right now is: " + ctime(time()) + ".\n");
    write("This object is " +
        (time() - object_time(this_object())) +
        " seconds old.\n");
```

There exists a convenient lfun `convtime` in the module `/lib/time` that will convert the timestamp to days, hours, minutes and seconds, excluding those entries which doesn't contain anything. Nice for more condensed listings.

Array/string conversion

[explode, implode]

Very often you come to situations where you either have a string that you would like to break up into smaller strings based on a regular substring, or conversly where you have a number of strings you would like to paste together to make up one single string. For this purpose you can use the efuns `explode()` and `implode()`.

The efun `explode()` takes two two strings as arguments, the first is the string you want to break up, and the second is the pattern that `explode()` looks for as a marker of where to break the string. It returns an array of strings holding the result. The efun `implode()` takes an array and a string as arguemnts, returning one string made up from the contents of the array with the string argument pasted in between all elements.

```
string *explode(string str, string expstr)
string implode(string *strlist, string impstr)
e.g.
    string fruit = "apple and banana and pear " +
                   "and orange and fatty eating it all";
    string *fruit_list;

    fruit_list = explode(fruit, " and ");
    dump_array(fruit_list);

    /* The result is:
        (Array)
        [0] = (string) "apple"
```

```
        [1] = (string) "banana"
        [2] = (string) "pear"
        [3] = (string) "orange"
        [4] = (string) "fatty eating it all"
     */

     fruit = implode(fruit_list, ", ");
     write(fruit + "\n");

    // The result is:
    // apple, banana, pear, orange, fatty eating it all
```

Array functions

[allocate, member_array, sizeof, pointerp]

Arrays are actually not arrays, but rather ordered lists of LPC data types. They can be made to contain any type of data, including other arrays. Keep in mind that arrays unlike other data types are copied by *reference* rather than by *value*. This means that when you assign an array to variable you do ***not*** copy the array, you merely store a reference, a pointer to the array, in the variable.

```
e.g.
    string *arr1, *arr2;

    arr1 = ({ 1, 2, 3 });
    arr2 = arr1;

    arr2[1] = 5;

    dump_array(arr1);
    /*
     * The output is:
     *
     * (Array)
     * [0] = (int) 1
     * [1] = (int) 5
     * [2] = (int) 3
     */
```

So, as you can see, changing the array `arr2' effectively changes the contents of `arr1' as well. You need to make a copy of `arr1' first, to make it unique. For example by simply adding an empty array `({})' to it.

As you have learnt arrays can be automatically allocated simply by writing them in the code, by adding elements to them or adding arrays to each other. However, if you need to allocate an array immediately to a specified size, you can use the `allocate()` efun. It takes as an argument the size of the array you want and initializes all elements, regardless of array type, to 0.

```
mixed *allocate(int num)
e.g.
    string *str_arr;

    str_arr = allocate(3);
    str_arr[1] = "Fatty is a flabby blimp";
    dump_array(str_arr);

    /* The result is:
```

```
        (Array)
        [0] = (int) 0
        [1] = (string) "Fatty is a flabby blimp"
        [2] = (int) 0
     */
```

If you need to find out if a particular item is a member of an array or the index of that item, you use the efun `member_array()`. It takes as arguments an array and an item of any type, returning the index if it is part of the array and −1 if it isn't. If several instances of the searched for item exists in the array, the first index is returned.

```
int member_array(mixed arr, mixed elem)
e.g.
    int *arr = ({ 1, 55443, 123, -3, 5, 828, 120398, 5, 12 });
    int index;

    // Replace all instances of the value '5' with '33'
    while ((index = member_array(arr, 5)) >= 0)
        arr[index] = 33;
```

A very important efun to use with arrays is `sizeof()`. It simply returns the size, the number of elements, in an array. It's very common that you need to loop through all elements of an array to do something, or perhaps just find the last element, and then you need to know the size.

```
int sizeof(mixed arr)
e.g.
    string *arr = ({ "Fatty", "the", "blurp" });

    write(implode(arr, " ") + " is wrong.\n");

    // Replace the _last_ argument, but remember that
    // in LPC we start counting at 0 so subtract 1.
    arr[sizeof(arr) - 1] = "blimp";

    write(implode(arr, " ") + " is correct.\n");
```

The efun `pointerp()` can be used to determine if a variable contains an array (of any type) or not. This is useful when you have a function that might return 0 (NULL value) if something goes wrong.

```
int pointerp(mixed arr)
e.g.

    string *arr;

    if (pointerp((arr = find_player("fatty")->get_blimps())))
        write("Fatty's blimps right now are: " + implode(arr, ", ") + ".\n");
    else
        write("Fatty doesn't have any blimps, stupid. He is one.\n");
```

Mapping functions

[mkmapping, mappingp, m_sizeof, m_indices, m_values, m_restore_object, m_save_object]

Mappings, as stated earlier, are lists of associated indices and values. A value is associated with another, so that by indexing with the first value you retrieve the second. Internally they are set up as hashed lists, which makes for very quick lookups. However, they are memory hogs, using up lots of memory as compared with

arrays.

How to allocate mappings then? Well, that's very easy. Just declare it and assign a value to an index value. If it exists, the old value is removed and the new put in its place. If it doesn't exist it is allocated and stored in the mapping automatically. You can also use two arrays, one with the indices and one with the values and combine those into a mapping with the efun `mkmapping()`. Just remember that the two arrays *must* be of the same size.

```
mapping mkmapping(mixed indarr, mixed valarr)
e.g.
    string *ind_arr, *val_arr;
    mapping mp;

    mp["fatty"] = "blimp";
    mp["mrpr"] = "unique";
    mp["olorin"] = "bloodshot";

    // Is the same as...

    ind_arr = ({ "fatty", "mrpr", "olorin" });
    val_arr = ({ "blimp", "unique", "bloodshot" });
    mp = mkmapping(ind_arr, val_arr);

    // You can give the arrays directly, instead of through
    // variables, of course.
```

As with arrays, there's a function available to determine if a given variable contains a mapping or not, `mappingp()`, that works in the exact same way. Use it in the same circumstances, i.e. typically when a function might or might not return a mapping and you need to know for certain that it contains a valid value before you try to index it.

To find the size of mapping you have to use the special efun `m_sizeof()`. However, it works exactly like the corresponding array function, returning the amount of elements in the mapping.

Removing elements from a mapping is slightly more complicated than with arrays however, you have to use the special function `m_delete()` to do that. `m_delete()` doesn't exactly remove an element, what it does is that it creates a new mapping that is a copy of the indicated mapping, apart from a particular value pair. As you can see, it takes the mapping to delete from and the index to the value pair you want removed as arguments:

```
mapping m_delete(mapping delmap, mixed elem)
e.g.
    mapping mp, mdel;

    mp["fatty"] = "blimp";
    mp["mrpr"] = "unique";
    mp["olorin"] = "bloodshot";

    mdel = m_delete(mp, "olorin");
    dump_array(mdel);

    /* Output:
     *
     *  (Mapping) ([
     *    "mrpr":"unique"
     *    "fatty":"blimp"
     *  ])
```

```
     */
```

Well... how to access all elements of a mapping then? Particularly one would want some kind of reverse function to `mkmapping()` earlier. Actually, there's two: `m_indices()` and `m_values()` which returns an array containing the index and value part of the given mapping respectively. Due to a linguistic confusion, the efun `m_indices()` has a double called `m_indexes()`. They both do the same thing however (actually just two names for the same function) so you can use either, as your linguistic preferences dictate. :)

However, now we come to a sensitive subject − order in mappings. As explained earlier a mapping has no defined internal order. Well... it has, but no order that you need or should worry about. This order also changes when you remove or add value pairs to a mapping. All in all this means that if you extract the indices and the values from a mapping, those two arrays *will* correspond to each other, the first index value corresponding to the first array value, *only* as long as the mapping hasn't been changed in between those two operations.

```
mixed m_indices(mapping mapp);
mixed m_values(mapping mapp);
e.g.
    // This function displays a mapping and its contents
    void
    dump_mapping(mapping mp)
    {
        int i, sz;
        mixed ind, val;

        ind = m_indices(mp);
        val = m_values(mp);

        sz = sizeof(ind);

        for (i = 0 ; i < sz ; i++)
            write(sprintf("%O", ind[i]) + " corresponds to " +
                  sprintf("%O", val[i]) + "\n");
    }

    /* Example run: dump_mapping(([ "fatty" : "blimp",
     *                              "mrpr" : "unique",
     *                              "olorin" : "bloodshot" ]));
     *
     * "olorin" corresponds to "bloodshot"
     * "fatty" corresponds to "blimp"
     * "mrpr" corresponds to "unique"
     *
     */
```

There are two functions that faciliates the saving and restoration of object data, `m_save_object()` will create a mapping that contains all global non−static variables, with the variable names as string indices corresponding to the actual values. You can then either save this mapping to file directly, or pass it to another function as you please. The converse of this efun; `m_restore_object()` takes a mapping as argument and reads the contents into the corresponding non−static global variables.

Type conversion

[atoi, atof, ftoa, itof, ftoi, str2val, val2str, sscanf]

Most user input is in the form of strings; you type something and then the game is supposed to act upon your answer. This calls for functions that can analyze your input and translate it to values you can use in your program. The syntactical analysis is *very* complicated, to say the least, and I'm going to leave that part for chapter three. However, let's look a bit at the value transformation bit. As stated input is in the form of strings, this makes it very interesting to convert strings to integers and floats, and vice versa.

Let's start with integers. Suppose you have received a string holding a numerical value and you want to use it computations. In order to convert it to the proper integer value you use the efun `atoi()`, very simply. It takes a string as argument and converts it to the corresponding integer. However, if the string contained non−numerical characters apart from leading or trailing spaces, 0 will be returned.

The name `atoi()` is derived from 'a(scii) to i(nteger)', for those of you who are interested to know.

```
int atoi(string str)
e.g.
    int val;

    val = atoi("23");

    write("23 + 3 = " + (val + 3) + "\n");
```

Floats have a corresponding efun, `atof()`, which converts a string to float. As you know by now, floats can't be converted to strings the same way integers can by simply adding them to another string, but require some other kind of treatment. The efun `ftoa()` will convert a float to a string, and the reverse functoin `atof()` will turn a string to a float, provided it contains a floating point number. Again, if the string contains any non−numerical characters the result will be 0.

For conversion between integer and float you have the efuns `itof()` and `ftoi()`. Just keep in mind that when you convert a float to integer, the decimal part will be cut off, not rounded.

There are many occations when you would want to store a value as a string and later convert it back to a value. For this purpose you have the two efuns `val2str()` and `str2val()`. The output from `val2str()` can be printed, but is not intended to. You can store any kind of variable contents as a string using these efuns.

The most used data converter, however, is the efun `sscanf()`. With `sscanf()` you can specify a pattern that should be scanned for a certain value, extract that and put it into a variable. This makes `sscanf()` a bit special since it handles variables given as arguments, so it's impossible to get the function address of `sscanf()`. I know this sounds pretty much like garbled greek to you at this moment, but trust me. I'll explain more in detail in chapter 3. Anyway, otherwise `sscanf()` is fairly simple; you provide a string to to search through, a pattern and the variables it should store data in, and it returns the number of matches it actually managed to make.

The string you give for pattern matching is interpreted literally apart from these control strings:

`%d'
: matches an integer number.

`%s'
: matches a character string.

`%f'
: matches a float.

`%%'

matches a %−character.

```
int sscanf(string str, string pattern, ...);
e.g.
    int wide;
    float weight;
    string orgstr;
    string wide_type, weight_type;

    /*
     * Assume the question "How wide and heavy do you think Fatty is?"
     * has been posed and answered to. Furthermore, assume that the
     * answer is given on the form '<amount> <sort> and <amount> <sort>',
     * as for example '4 yards and 3.2 tons'. Assume the first value
     * always is an integer and that the second is a float.
     *
     * Assume that this answer is given in the variable 'orgstr'
     *
     * The above is naturally only convenient assumptions to make the
     * example easy to write. In reality you'd better be prepared for
     * any kind of format being given as answer.
     */

    if (sscanf(orgstr, "%d %s and %f %s", wide, wide_type,
               weight, weight_type) != 4)
    {
        write("Please give a full answer!\n");
        return;
    }

    write("Aha, you think Fatty is " + wide + " " + wide_type +
          " wide and " + ftoa(weight) + " " + weight_type + " heavy.\n");
```

Math functions

[random, rnd, sin, cos, tan, asin, acos, atan, atan2, exp, log, pow, sinh, cosh, tanh, asinh, acosh, atanh, abs, fact, sqrt]

The efun `random()` will return an integer random number from 0 to one less the number you give as an argument. For example `random(8)` will return an integer from 0 to 7.

The rest of the mathematical functions all return floats and take floats as arguments. The trigonometric functions use radians, not degrees. Keep this in mind.

`float rnd()'
: Returns a random number in the range 0 (inclusive) to 1 (exclusive), i.e. it might be 0, but never 1.

`float sin(float)'
: Compute the sinus value of an angle.

`float cos(float)'
: Compute the cosinus value of an angle.

`float tan(float)'
: Compute the tangens value of an angle.

`float asin(float)'
: Compute the arcus sinus value in the range −pi/2 to pi/2.

`float acos(float)'
: Compute the arcus cosinus value in the range 0 to pi.

`*float atan(float)*'
: Compute the arcus tangens value in the range −pi/2 to pi/2.

`*float atan2(float x, float y)*'
: Compute the argument (phase) of a rectangular coordinate in the range −pi to pi.

`*float exp(float)*'
: Compute the exponential function using the natural logarithm e as base.

`*float log(float)*'
: Compute the natural logaritm.

`*float sinh(float)*'
: Compute the sinus hyperbolicus value.

`*float cosh(float)*'
: Compute the cosinus hyperbolicus value.

`*float tanh(float)*'
: Compute the tangens hyperbolicus value.

`*float asinh(float)*'
: Compute the arcus sinus hyperbolicus value.

`*float acosh(float)*'
: Compute the arcus cosinus hyperbolicus value.

`*float atanh(float)*'
: Compute the arcus tangens hyperbolicus value.

`*float abs(float)*'
: Compute the absolute value of the given argument.

`*float fact(float)*'
: Compute the factorial (gamma function) of the given argument.

`*float sqrt(float)*'
: Compute the square root of the given argument.

File handling

[save_object, restore_object, save_map, restore_map, write_bytes, read_bytes, write_file, read_file, file_size, file_time, rename, rm, ed]

Using files for storage of data is very important. Followingly there are a number of functions available to aid you with this. However, let me start with a little sermon on the subject of CPU usage:

Reading and writing to files is very CPU intensive, perhaps not in the respect that the CPU actually has a lot to do while it happens but that it is unable to do anything else at the same time. In other words, reading and writing large portions of data often will slow the game down significantly. To impose a small limit on excessive usage of memory, disk and CPU, it's impossible to handle more than ca 50 kb of data at one time. Files may be bigger, but you can't write or read bigger chunks than that. This means you have to split up work on big files into portions to be executed sequentially, preferrably with a pause between each execution to give the rest of the game time to do something. So, please keep in mind that this limit isn't there to annoy you, to be sidestepped by nifty code, but as a reminder that you are hogging the resources and should let others do something as well. Amen.

Let's start with the very basic conept of storing and restoring objects. What you want to do usually is to store the global variables to file, pending later restoration. For this purpose you use the efuns `save_object()` and `restore_object()`. They both take a filepath as argument and naturally have to specify a file which the object in question is privileged to write or read, respectively. The resulting savefile will have a name ending in '.o', and you must remember to specify this extension to `restore_object()`. This is optional

with `save_object()` since it's added automatically if you forget it. `restore_object()` returns the integer 1 on successful reading of a file, and 0 otherwise. The contents of the saved file are a list of all global variables with their contents on the same line separaterd by a space. The storage format of the string is the same as with `val2str()` mentioned earlier for the content of a single variable. naturally `save_object()` will store the names of the variables as well in front of the data it contains.

An important concept to remember is that data files stored with `save_object()` are text files, and hence editable with the internal `ed()` editor. However, the lines might become very long if you store large arrays for exampe. `ed()` will then truncate the lines at the maximum length, and if you then store the contents back to file you will in fact destroy part of the data, making it impossible to read back. This unfortunately is a very common mistakes with new archwizards who want to hack the `KEEPERSAVE.o` file manually, instead of going through the commands supplied for that purpose.

Mappings are the most convenient data type to be used with saving variables. Just store the data you want in a mapping with a string describing it as index, then store the mapping with the efun `save_map()` for later restoration with `restore_map()`. The advantage with this method over `save/restore_object()` is that you aren't limited to global non−static variables but can store whatever you like. The drawback is that retrieving data is a bit more complicated.

```
void save_object(string savepath);
int restore_object(string readpath);
void save_map(mapping mapp, string savepath);
mapping restore_map(string readpath);
e.g.
    /*
     * Assume these global variable definitions:
     *
     * string name, *desc;
     * int flip;
     * mapping data_map, smap;
     *
     * Assume we are interested in storing name, desc, flip and data_map
     */

    // Set object inherent privileges by giving it the euid of the
    // creator of the file
    setuid();
    seteuid(getuid());

    // Method 1 save
    save_object("myfile");

    // Method 1 restore
    if (restore_object("myfile"))
        write("Yes!\n");
    else
        write("Naaaah..\n");

    // Method 2 save
    smap = ([ "name" : name,
              "desc" : desc,
              "flip" : flip,
              "dmap" : data_map ]);
    save_map(smap, "myfile");

    // Method 2 restore
    smap = restore_map("myfile");
    if (m_sizeof(smap))
```

```
    {
        name = smap["name"];        // Restore name
        desc = smap["desc"];        // Restore desc
        flip = smap["flip"];        // Restore flip
        data_map = smap["dmap"];    // Restore data_map
        write("Yes!\n");
    }
    else
        write("Naaaah..\n");
```

A fact to be remembered is that the save format used internally by `save_object()` and `save_map()` is the same, which makes it both possible and sometimes very useful to restore data from objects that have saved their contents with `save_object()` by using `restore_map()` and then just picking out the pieces you want from the resulting mapping. Assume that you only would have been interested in restoring the variable 'desc' in the example above, then you never would have bothered with the other statements in the Method 2 restore. Beware that using `restore_object()` on a savefile stored with `save_map()` requires the indices used in the original mapping to have the same name as the global variables intended to receive the data, something that doesn't have to be true, as exemplified above. Restoring the Method 2 savefile with Method 1 restore will not result in an error, but it will fail to restore the variable 'data_map'.

Well, these are all methods for storing data in variables. Very often you want to store free−form data however, and not just data in variables. For this purpose you can use the efuns `write_bytes()` and `read_bytes()`, or `write_file()` and `read_file()`. Basically both pairs of functions do the same thing, i.e. save or read a string of certain length from file. The only difference is that `write_bytes()` can be used to overwrite a portion of a file, while `write_file()` only can append to a file. Also, `read_bytes()` acts on exact bytes, while `read_file()` acts on lines separated by newlines. Both write functions return 1 on success and 0 on failure. Both read functions return a string with the result of the read operation on success, on failure they return 0, so check the result with `stringp()` to make sure it has succeeded.

```
int write_bytes(string path, int pos, string text);
string read_bytes(string path, void|int pos, void|int num);
int write_file(string path, string text);
string read_file(string path, void|int pos, void|int num);
```

You can get information about a file as well. You get the size of the contents with the efun `file_size()`, but it can also be used to check the type and existence of a file. If the returned number is positive, it is a file and the number represents the size in bytes of the contents, if the file doesn't exist, it returns −1 and if the file actually is a directory it returns −2. To get the time of last modification you use the efun `file_time()`.

```
int file_size(string path);
int file_time(string path);
e.g.
    void file_info(string path)
    {
        int type, tm;

        type = file_size(path);
        tm = file_time(path);

        write("The file '" + path + "' ");
        switch (type)
        {
        case −1:
            write("doesn't exist.\n");
            break;
```

```
        case -2:
            write("is a directory, last modified " + ctime(tm) + ".\n");
            break;

        default:
            write("is " + type + " bytes in size, last modified " +
                  ctime(tm) + ".\n");
            break;
        }
    }
```

If you want to rename or move a file you can use the efun `rename()`. Beware that this operation actually first copies the file and then removes the old one. It can also be used to move directories. If you wish to remove a file entirely, you use the efun `rm()`. The efun `rm()` returns 1 on success and 0 on failure, however beware that `rename()` works just the opposit way around, it return 1 on *failure* and 0 if all is well.

```
int rename(string oldpath, string newpath);
int rm(string path);
e.g.
    if (rm("myfile"))
        write("Ok, removed.\n");
    else
        write("Sorry, no go.\n");

    if (rename("myfile", "yourfile"))
        write("Nope, still the same...\n");
    else
       write("Ok!\n");
```

The internal editor 'ed' is actually an efun that operates on a file. You can use it for whatever purpose you like, but keep in mind that most people don't really know how to use it. Also remember that the efun `ed()` can be used to create new files and edit old as per the privileges defined by the object. You can provide a function pointer to a function that will be called on termination of the efun. If you don't provide a filepath, the user will be expected to give the path and name of the file from within the editor.

```
void ed(void|string path, void|function exit_func);
```

Directory handling

[mkdir, rename, rmdir, get_dir]

Creating, renaming and removing directories are handled by the efuns `mkdir()`, `rename()` and `rmdir()`. You need write permissions in the directory you are doing this, of course. `mkdir()` and `rmdir()` return 1 on success and 0 on failure. `rename()`, as already pointed out, works the other way around and returns 1 on failure, 0 on success. `rmdir()` only works if the directory you want to remove is empty, i.e. contains no other files or directories.

```
int mkdir(string path);
int rename(string oldpath, string newpath);
int rmdir(string path);
```

For listing the contents of a directory, you can use the efun `get_dir()`. It simply returns an array with the names of all files in the specified directory, or an empty array on failure.

```
string *get_dir(string path);
```

```
e.g.
    string *dir_contents;
    int i, sz;

    dir_contents = get_dir("/d/Domain/fatty");

    for (i = 0, sz = sizeof(dir_contents) ; i < sz ; i++)
    {
        // See the code for file_info in a previous example
        file_info(dir_contents[i]);
    }
```

Screen input/output

[write, write_socket, cat, tail, input_to]

By now you're fairly familiar with the efun `write()`, it simply outputs data to whoever is registered as listening, it might be a player or it might be an object. Usually this function suffices, you have full control of what you want to write and who you want to write it to. However, there exists one function that sometimes is necessary, namely `write_sockect()` that *only* writes to the interactive user. If none exists, it writes to the central error log instead. It works analogous to `write()` in all other aspects. Coding ordinary objects you will never need to use this one, it's mostly or perhaps I should say only, used for certain mudlib objects, particularly to do with logging in players.

Writing is nice, but sometimes you want to relate whole parts of files quickly. Then you should use the efun `cat()`. It will print a specified portion of a file directly on the screen quickly and easily. There even exists a special efun called `tail()` for listing only about the last 1080 bytes of a file in the same manner. `cat()` makes sure that it starts reading from a new line and returns the number of lines actually read. `tail()` returns 1 on success and 0 on failure.

```
int cat(string path, int start, int len);
int tail(string path);
e.g.
    // List 80 lines in the file TESTFILE, 20 lines down
    cat("TESTFILE", 20, 80)

    // List the ending of the same file
    tail("TESTFILE);
```

A small warning, if you use `cat()` on long files you might get an eval−cost error. This is fairly common when you have logs or instructions you want to display, and forget to cut them up into smaller parts.

Most input to LPC programs comes as arguments to commands. However, at times you need to actually ask the player for input and he needs to answer. This poses a special problem since object execution in the gamedriver is sequential; if you stop to wait for an answer, all of the game will stop along with you while the player makes up his mind (if ever) and types. This obviously won't do. Instead you can use the special efun `input_to()` which allows you to specify a function which then will be called with whatever the player types as argument, after the completion of the current function. This sounds complicated but is not, just look at this example:

```
void input_to(function func, void|int noecho);
e.g.
    interrogate_fun()
    {
        write("Please state your name: ");
```

```
    input_to(func_2);
}

func_2(string n_inp)
{
    string name;

    if (!strlen(n_inp))
    {
        interrogate_fun();
        return;
    }

    name = n_inp;
    write("\nState your sex (male or female): ");
    input_to(&func_3(, name));
}

func_3(string s_inp, string name)
{
    string sex;

    if (s_inp != "male" && s_inp != "female")
    {
        write("\nState your sex (male or female): ");
        input_to(&func_3(,name));
        return;
    }

    sex = s_inp;
    write("\nState your occupation: ");
    input_to(&func_4(, name, sex));
}

func_4(string o_inp, string name, string sex)
{
    string occupation;

    if (!strlen(o_inp))
    {
        interrogate_fun();
        return;
    }
    occupation = o_inp;
    write("\nYour name is " + name + ",\n"
          + "you are a " + sex + " " + occupation + ".\n"
          + "\nThank you for your cooperation!\n");
}
```

If you specify the second argument to `input_to()` as 1, whatever the player types will not be echoed on his screen. This is what you want to do for passwords and other sensitive information.

Advanced LPC and Mudlib

This chapter will deal with the more advanced portions of LPC. This does *not* mean that you are able to do without them, other than in very simple objects. It's just that you *really* need to understand the basics of LPC as described in the earlier two chapters before you can assimilate what I'll describe here.

Please don't hesitate to review earlier portions as necessary while you read the text.

Function data type, part 3

The function type hasn't been discussed in any depth earlier. You've seen it used but not really explained. However, it's very important that you do understand how they work, as you can build very complex and effective expressions using them, and above all you will bot see them used and be forced to use them here and there. All the functions that used to take function names as strings now take function pointers as arguments instead.

The basics of the function type

Functions are accessed through function pointers. As has been demonstrated implicitly earlier, every function call basically is nothing but a dereferenced function pointer along with a list of arguments. Take this simple example:

```
void
my_func(string str, int value)
{
    write("The string is '" + str + "' and the value is " + value + ".\n");
    return;
}
```

The function is then called, as demonstrated earlier, by giving the function name followed by a list of the arguments within brackets:

```
e.g.
    my_func("smurf", 1000);
```

Now we add the idea of the function type where the address of the function can be taken and assigned to another variable:

```
e.g.
    function new_func;

    new_func = my_func;       // or equivalent
```

```
    new_func = &my_func();

    my_func("smurf", 1000);   // or equivalent
    new_func("smurf", 1000);
```

Beware that before the variable `new_func` has been set to a proper value, it doesn't refer to any function at all. Using it will cause a run−time error.

Partial argument lists

As seen in the previous chapter, it's possible to assign the reference of a function to a function variable with another name. Another handy feature is that it's possible to set this new function up so that it defines a number of the arguments to the original function as constants, leaving only the remaining as variables. Look at this example:

```
e.g.
    void
    tell_player(object player, string mess)
    {
        player->catch_msg(mess + "\n");
    }

    void
    my_func()
    {
        function mod_tell;

        mod_tell = &tell_player(this_player(), );

        mod_tell("Hello!"); // Equivalent to

        tell_player(this_player(), "Hello!");
    }
```

This works fine for any number of arguments. The remaining arguments will be filled in left to right, just as you would like to expect. i.e. a function with the header `void func(int a, int b, int c, int d, int e)` defined as `function my_func = func(1, , 3, 4,)` and called as `my_func(100, 101)` is eqvivalent to the call `func(1, 100, 3, 4, 101)`.

Complex function applications

Writing efficient code

Actually, this subject is closely linked to the very thing I said I would *not* explain − namely how to program. I'll retract my words − a bit − and talk about some 'what to do's and even more importantly 'what *not* to do's.

Efficient loops

This might seem rather trivial, in how many ways can you mess up a loop anyway? Well... actually, quite a few. Let's start with the most common mistake. Assume you have a large array, let's call it 'big_arr', and let's

assume you want to loop over all elements in that array, what do you do? "Simple!", you exclaim, "A 'for' loop, of course!". Sure... but how do you write it? Well, the most common implementation usually looks something like this:

```
int i;

for (i = 0 ; i < sizeof(big_arr) ; i++)
{
    // Here we do something with the array
}
```

Ok... now why is this bad? Well, if you review the chapter on how the `for` statement works, you'll find that the three parts inside the round brackets actually gets executed. The first one once at the start, the second (or middle) part *every* time the loop is run and the third part also every time the current loop is finished.

This obviously means that the `sizeof()` function gets executed every time the loop is run. rather a waste of time given the fact that the array hardly is intended to change size. If it was, that would have been another matter, but as it isn't... No. Write like this instead:

```
int i, sz;

for (i = 0, sz = sizeof(big_arr) ; i < sz ; i++)
{
    // Here we do something with the array
}
```

See? The variables 'i' and 'sz' gets assigned their respective values at the start of the loop, and only then. The counter 'i' gets set to 0 and the size variable 'sz' gets set to the size of the array. During the entire loop after that, 'i' is compared with 'sz' instead of repeatedly recompute the size of the array.

Believe it or not, this is a very common mistake, all people do it. While the savings in doing as I suggest might not seem that great, well... multiply this small gain in one loop by all the loops in the mud and all the number of times that those loops are run and you'll end up with quite a big number. The added cost in doing this is one local variable, which is a small enough price to pay.

Keep this general problem in mind since a lot of cases don't use arrays perhaps, but mappings or other general containers of items you want to loop through. The solution apart from specifics in identifying the size of that container is always the same.

Abusing defines

A common mistake is to put ***HUGE*** arrays and mappings in a define. It's very tempting really, assume for example that you have a mapping that contains the definitions of guild ranks, descriptions, various skill limits, benefit adjustors etc in one big mapping with the rank as index. Very often you'd then need to index that mapping to look up things. Probably it'll be done dozens of times in the central guild object. You'd have something like this:

```
// Top of file

#define GUILD_MAP ([ 0: ({ "beginner", "Utter newbie", 3, 2343, ... }), \
                     1: ({ ....                                          \
                     ... /* Perhaps another 10-20 lines or more */       \
                   ])
```

```
// code, example of use
        write("Your rank is: " + GUILD_MAP[rank][1] + "\n");
// more code...
```

However... just pause for a second and consider what the `#define` statement really does... well, it *substitutes* whatever you had as a pattern for the `#define` body. So, in every instance where you had written `GUILD_MAP` the entire mapping would be copied in. And every time it was put in, the gamedriver would have to interprete, store and index the mapping again. It doesn't take a genious level of intelligence to realize that this is a horrible waste of both memory and time.

So... instead of doing it this way you store the mapping in a global variable. Then you use that variable as you use the define. I.e.

```
// Top of file

mapping GuildMap;

create_object()
{
    // code

    GuildMap =  ([ 0: ({ "beginner", "Utter newbie", 3, 2343, ... }), \
                   1: ({ ....                                          \
                   ... /* Perhaps another 10-20 lines or more */       \
                ]);
}

// code, example of use
        write("Your rank is: " + GuildMap[rank][1] + "\n");
// more code...
```

Traps and pitfalls

It's easy to get confused, to do things that look good but later prove to be pure disaster. This chapter will deal with some of the more common mistakes you're likely to make.

Most of the stuff here has been mentioned before while explaining the functions that are involved. However, judging by the amount of mistakes that being made all the time, it won't hurt to rub it in once more.

- Mapping/Array security: How to make mappings and arrays secure
- Alarm loops: Crashing the game with alarms

Mapping/Array security

The problem, as indicated in chapter 2.2.9 ealier, is that mappings and arrays aren't copied every time they are moved around. Instead only a reference is passed. This is the basis for a lot of security blunders in the code. Consider this example where the object is a guild object that handles the membership of a guild. The global string `Council` which is saved elsewhere using `save_object()` contains the list of guild members.

```
string *Council;

public string
query_council()
{
```

```
    return Council;
}
```

This looks all right, but... in fact you return the pointer to the original array. If someone else wants to add a member to your guild council he only has to do this:

```
void
my_fix()
{
    string *stolen_council;

    stolen_council = YOUR_GUILD_OB->query_council();

    stolen_council += ({ "olorin" }); // Add Olorin to the council
}
```

How to fix this then? Well, simply modify your `query_council()` routine to return `Council + ({})` instead, and all is well. Easy to miss, but... sooooo important!

Alarm loops

Look at this function:

```
public void
my_alarm_func(int generation)
{
    set_alarm(1.0, 1.0, my_alarm_func(generation + 1));
    tell_object(find_player("<your name>"), "Yohooo! " + generation + "\n");
}
```

What's happening here? Well, every second an alarm is generated, calling itself in one second. What does this mean? Let's look at the development of alarm calls after the first call:

```
1 second:
    Yohoo! 0 (original call)
2 seconds:
    Yohoo! 1 (repeat of 1 sec 0)
    Yohoo! 1 (new from 1 sec 0)
3 seconds:
    Yohoo! 1 (repeat of 1 sec 0)
    Yohoo! 2 (repeat of 2 sec 1)
    Yohoo! 2 (repeat of 2 sec 1)
    Yohoo! 2 (new from 2 sec 1)
    Yohoo! 2 (new from 2 sec 1)
4 seconds:
    Yohoo! 1 (repeat of 1 sec 0)
    Yohoo! 2 (new from 3 sec 1)
    Yohoo! 2 (repeat of 2 sec 1)
    Yohoo! 2 (repeat of 2 sec 1)
    Yohoo! 3 (repeat of 3 sec 2)
    Yohoo! 3 (repeat of 3 sec 2)
    Yohoo! 3 (repeat of 3 sec 2)
    Yohoo! 3 (repeat of 3 sec 2)
    Yohoo! 3 (new from 3 sec 2)
    Yohoo! 3 (new from 3 sec 2)
    Yohoo! 3 (new from 3 sec 2)
    Yohoo! 3 (new from 3 sec 2)
```

```
... etc.
```

As you can see we have exponential growth here... VERY funny... the game will grind to a halt practically at once! This, if you didn't know it, is so stupid as to be a demoting offense. Is this easy to do by mistake? Well... I've seen it a few times. Often enough to warrant this warning anyway. Oh, try this on Genesis and you're dead meat! Consider yourself warned!

LPC Index

#

'

;

?

a

b

c

e

i

m

n

p

s

t

v

w

Efun/Sfun Index

a

b

c

d

e

f

g

i

l

m

- m_save_object() − Save an object in a mapping
- m_sizeof() − Find the size of a mapping
- m_values() − Return a list with the value part of a mapping
- mappingp() − Determine if a variable is of type mapping, mappingp() − Determine if a variable is of type mapping
- member_array() − Find if an element is part of an array
- mkdir() − Create a directory
- mkmapping() − Create a mapping
- move_object() − Move an object to the inventory of another object

n

- notify_fail() − Store a fail−message string

o

- object_clones() − Find all clones of a given object
- object_time() − Get creation time for an object
- objectp() − Determine if a variable is of type object

p

- pointerp() − Determine if a variable is of type array, pointerp() − Determine if a variable is of type array
- present() − Determine if a named object resides in the inventory
- previous_object() − Obtain a pointer to the previous object

q

- query_verb() − Get the last given command word

r

- random() − Get a random number
- read_bytes() − Read text from file by byte
- read_file() − Read text from file by line
- remove_alarm() − Remove a previously set alarm
- rename() − Rename or move a directory
- rename() − Rename or move a file
- restore_map() − Restore a mapping from file
- restore_object() − Restore an object's variables from file
- rm() − Remove a file
- rmdir() − Remove a directory
- rnd() − Get a random floating point value between 0 and 1

s

t

u

v

w

Lfun/Macro Index

c

e

i

l

m

r

Type Index

a

f

i

m

o

s

v