M

N

O

P

Q

R

S

T

U

V

W

For example, you want to find the user id of an object, but you can't remember exactly what the function was called. You have a pretty good notion it ended with the letters 'id' though:

```
> man -k *id

--- simul_efun:
export_uid  geteuid     getuid      seteuid     setuid
```

The output tells you that the matches can be found in the chapter *simul_efun* of the manual. Had their been matches in other chapters, they would have been displayed as well.

Now you also know that the function was 'getuid', so you just type `man getuid` (or even `man simul_efun getuid` if you want to be very specific) to get the full text.

The really experienced LPC coder has a good grasp of which of the objects provided in the mudlib does what, so that he'll know which object to inherit for a specific job. He is also rather at home with what the function-sets each object defines, but naturally he doesn't know all of them. What follows from this is that any coder, experienced or not, simply *has* to spend quite a bit of time reading the objects to find out exactly how to use them when he needs them.

A great help in this respect is the *sman* command. The source code manual simply is a collection of all headers of all functions in */cmd*, */lib*, */obj*, */secure*, */std* and */sys*.

The command allows you to search for partial names as well, so finding the exact function you need usually is a matter how having just an inkling of what you're after, then spending some time with *sman* to find the exact command.

For example, you need to find the function in a player that returns his occupational guild name, but you don't know which object defines it, nor what the function was called.

```
> sman -k *guild*
--- /lib/guild_support:
create_guild_support     init_guild_support

--- /secure/master:
add_guild_master                      query_guild_type
guild_command                         query_guild_type_int
guild_filter_type                     query_guild_type_long_string
guild_sort_styles                     query_guild_type_string
load_guild_defaults                   query_guilds
query_guild_domain                    remove_guild_master
query_guild_is_master                 set_guild_domain
query_guild_long_name                 set_guild_long_name
query_guild_masters                   set_guild_phase
query_guild_phase                     set_guild_style
query_guild_short_name                set_guild_type
query_guild_style

--- /std/guild/guild_base:
list_major_guilds                     query_guild_not_allow_join_guild
query_guild_keep_player               query_guild_skill_name
query_guild_leader                    query_guild_style
query_guild_member

--- /std/guild/guild_lay_sh:
query_guild_incognito_lay             query_guild_tax_lay
```

```
int s, x;
s = 3;
x = SQUARE(s++);  // You might _think_ x = 9 and s = 4 when this is done.
```

Well, like I said, the preprocessor happily substitues the pattern you specified, and what you end up with is:

```
int s, z;
s = 3;
x = s++ * s++; // This makes x = 12 and s = 5!
```

Variations on this theme makes #define a truly double-edged sword. So please use them as sparingly as possible for as simple tasks as possible. The basic rule is to keep macros short and fairly simple, however even then you can't be certain you won't be bitten somehow.

3.6 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.

3.7 Mapping/Array security

The problem, as indicated in chapter 2.2.9 earlier, 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!

NB! This pragma is slightly obsolete, as it is globally defined. It won't harm to put it into the program, but it won't do anything that isn't already done either.

'`#pragma no_shadow`'
This directive prevents an object from being shadowed using the shadow kfun.

3.5 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.

3.5.1 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.

object. If you create an object without inheriting /std/object.c or one of its derivates, that object can never be inserted in to the inventory of any other object, which in turn means that it never can appear in the game. Since the /std/object.c object contains a number of unmaskable functions, this means that a shadow as such never can be inserted into the game. Well, unless it inherits /std/object.c itself, and only shadows some kind of non-insertable object, of course. I suppose that this isn't impossible as such, just rather unlikely.

Assume that an object has been shadowed by several objects, and that the same function is shadowed in all objects, which shadow finally gets the call? This is not an impossible situation if you're thinking that I'm posing a trick question; since you're not shadowing a shadow, but the original object, and since the function is shadowable there's no preventing this from happening. The answer is that it's the last function to shadow the object that 'wins'. Obviously this is not a very good situation, so often the best way of making sure that your object is the only final object to shadow another is to define the function query_prevent_shadow() to return 1. However, this obviously makes it impossible for all other objects to shadow any other function as well, so one has to be careful lest one interrupts some other object from working.

By far the easiest way of creating a working shadow object is to inherit the standard shadow object /std/shadow.c and use the functionality provided there. With that module you're not only getting a standardized way of dealing with shadows that most other object-managing objects recognize and can use, you're also getting a number of handy functions that make sure that the shadow doesn't shadow the same object twice by mistake, that autoloading shadows on players automatically latch on to the player on login again as well as provide a mechanism for removing the shadow.

Assume for example that you want to redefine the gender of a player who's been affected by an evil spell of some kind. The function you want to shadow is called query_gender_string() and can be found in /std/living/gender.c. This is how you do it:

```
inherit "/std/shadow";

int
start(string pl_name)
{
    return(objectp(shadow_me(pl_name)));
}

int
stop()
{
    remove_shadow();
}

string
query_gender_string()
{
    return "xxx";
}
```

Please notice that I've added both a start and a stop function for easy testing; these really aren't necessary as long as the shadow somehow calls shadow_me() with the target (either as a name or an object reference) as argument.

The standard shadow module also provides easy access to the original object. The variable shadow_who is set to the original object, so in the shadow object any call on the form shadow_

Well now, that wasn't too difficult, now was it?

First, I loop through all users and look through their entire inventories. I filter out those users who are carrying a shield, and obtain their names. The resulting array I then sort, capitalize and paste together into a string with a `,` in between each name. Finally I return the resulting string.

In other words, I obtain a list of all users who are carrying a shield.

NB! Now it is time for a warning. As you can see from the previous example, I had no trouble at all generating a list of objects that probably reached into the thousands before starting to narrow things down. Let me check... Yes, in this case more than 1500 objects, all being called to see if they match the name `"shield"` and then other on top of that further operations applied to the resulting lists. Obviously it's very easy to generate massive calls which tax the computer quite a lot.

What this warning amounts to really is to think *before* you code, so that you won't bog down the computer with lots of calls that perhaps aren't strictly necessary.

3.2 How to parse player commands

Parsing commands written by a player can be one of the more taxing tasks in programming the mud. It's also one of the more important ones, as intelligent command parsing can turn an otherwise annoying object into a useful one.

However, instead of having to do command parsing on your own there's a very neat efun called `parse_command()` available that'll take care of most of the chore for you. It's slightly tricky to use at first, but learning it is most worthwhile.

As you'll find, `parse_command()` looks slightly similar to `sscanf()` operating on word basis. Like `sscanf()` it will return values in its arguments (pass by reference) which might look slightly confusing but is necessary in this case.

The syntax is:

```
int parse_command(string command, object target|*targets, string pattern,
                  mixed arg, ...)
```

Let's take this one bit at a time.

The command above simply is what the player wrote; the text you want to parse for its contents.

The target or list of targets is the objects you want the command to operate *on*. I.e. the object or objects in which you expect the player to find the items he is manipulating.

The pattern is the text pattern you wish to match with the command.

The arguments is a list of arguments where the results of successful parsing will end up.

The efun will return a boolean value (1 or 0) denoting whether a match was possible or not.

Let's look at an example here: Assume you have coded an apple tree and you want the user to be able to pick one or more apples from the tree, or from the room the tree resides in which might have fallen down.

In the tree, you would add this code:

```
static int do_pick(string arg);  // Forward declaration

init()
```

```
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)`.

As you now understand, the `&function_name(argument_list)`-way of obtaining function references always count as a partial function definition, even if the argument list is empty. This really isn't that important from a practical point of view, not from your end anyway, but it's good to know what you're talking about even so.

3.1.3 Complex function applications

With the previous chapters in memory, the interesting question becomes 'What can you put into one of these function calls?'. The answer is 'Practically anything!', it's just a matter of knowing how to write the code in the right way (as usual).

The tricky bit usually is operators. For most of them you use the special keyword *operator* to encapsulate the actual operator. The operators that can be encapsulated are +, -, *, /, %, &, |, ^, >>, <<, <, >, <=, >=, ==, !=, []. For the []-operator you then have to index it in order to make use of it, but more about that later.

Often enough you want to perform one operation and then send the result to another one. You do that with the `@`-keyword, but beware that execution is right-to-left on that line, and that you'll have to figure relational operators that way too!

This all sounds a bit difficult to understand, but let me give you an example to make everything clear:

Assume you have an array like this representing some player names:

```
string *arr = ({ "Bocephus", "Thurol", "Elessar", "Budwise",
                 "Akhan", "Logg" });
```

Now assume you want to filter out the names that are more than five characters long. Usually this involves running a loop over the contents of the array, examining each entry in turn to determine its invidual length, and finally building an array containing the result.

```
string *
```

A neat feature is that the last specified object automatically gets stored as *$*, which means that having once specified an object in any tracer command, you then only need to type `<Command> $ <options>` to specify the same object again.

2.5.13 Update, load, clone and replace an object

With this command, you can update an objects master object, reclone and move it to it's current location, then destruct the old instance if all went well.

The reload process tries to move objects 'nicely', however, if that fails, it will attempt to force move automatically.

If the reload process encounters a fatal error during execution, it will be aborted and you will continue to have the old object.

Syntax: `Reload <object spec>`

2.5.14 Tail the file associated with an object

This command works just like *Cat* and *More*, with the difference that it displays just the last few lines of a file.

`cpu` This parameter lists the cpu time consumed by the specified object. This information actually is only interesting to mudlib developers and only works if the driver is compiled with the PROFILE_OBJS flag.

`flags` This parameter lists all gamedriver-relevant object flags associated with the specified object together with some object status information. This information actually is only interesting to gamedriver developers.

`functions`
This parameter lists all function names defined by the specified object. In case the list looks short to you, remember that most objects consist of a long inheritance chain, and the inherited object's functions are not listed; you will have to ask for them separately.

`info` This parameter lists some basic gamedriver-relevant information associated with the specified object. This information actually is only interesting to gamedriver developers.

`inherits`
This parameter lists the entire inheritance chain of the specified object.

`inv | inventory`
This parameter lists the inventory of an object and numbers them for easy further referencing.

`items` This parameter lists all pseudo-items, i.e. items that only have a descriptive existence or command functionality existence, that has been added to an object. The list of command items is split in a section with commands and a section with items. Each pair of commands and items is terminated with their index number.

`light` This parameter lists the light status for the specified object and reports on the effects of all contained objects as well. It will tell you the current light status and whether the object generates or absorbs light.

`profile` This parameter lists any stored profiling information associated with the specified object. This information is only interesting to gamedriver developers and only works if the driver is compiled with the PROFILE_FUNS flag.

`props | properties`
This parameter lists all properties stored in the specified object.

`shadows` This parameter lists all shadow active on the referenced object.

`vars | variables`
This parameter lists all variables of an object.

`wizinfo` This parameter prints the special wizard information that can be stored in the OBJ_S_WIZINFO property in any object. This information is directed only at wizards and should give directions on handling or the purpose of the object.

Please notice that it's the responsability of the creator of the object to make sure that this information is added to the object.

2.5.7 Ed the file associated with an object

This command starts the *ed*-editor on the file referred to by the specified object.

Syntax: `Ed <object spec>`

'`path`' This specifies a file path to an object. If you want a specific indiviual item, you add the clone number. For instance not just `~Ansalon/guild/society/obj/nametag` but specifically exactly `~Ansalon/guild/society/obj/nametag#22144` and no other.

The non-specific path identifies the master object, of course.

'`@name`' This specifies the name of a player *or* a *living* (a monster) anywhere in the game. Please notice that the tracer will find the name as registerd by a call of the efun `set_living_name()` (described earlier) and not the name set by `set_name()` in the object itself. A player is automatically added to the list of *living* objects.

'`*name`' This specifies the name of a player, and nothing else.

'`here`' This specifies the environment of yourself, usually you are standing in a room of some kind.

'`me`' This specifies yourself, your player object.

'`#num`' This specifies the object number 'num' in the scope you indicate. If for example you *know* that the sword you are carrying is the third object, the reference would be '#3', simply. Please beware that object order may change without warning. If for example someone gives you something just before you type the command, object '#3' will no longer be the one you thought it was. Instead make extensive use of tracer variables (explained in the command *Set*) to void this problem.

'`$var`' This specifies the contents of a tracer variable (explained later in the command *Set*).

Objects often exist in some sort of environment. Sometimes in the same place as another object, sometimes inside an object, sometimes the object is enveloping another. In order to give a relation of the type "the second object inside the teddybear in the same room as I am standing in", you give a list of object specifications separated by a *:*. The environment of an object is specified with a caret (*:^*).

For example, the previous description would come out as `me:^:teddybear:#2`. This really isn't very complicated, you just build the path based on an object that you are certain you know which it is.

Another example: "the sword stored in a bag held by the player Adam": `*adam:bag:sword`.

2.5.1 Perform a command in the environment of a player

This command actually moves you to the environment of the specified player to perform the given command. The move is a forced move so that weight accounting and destination tests aren't performed, but you are still actually *moved* there, so any triggers on entering and leaving will be notified.

Syntax: `At <player name> <command>`

Usually there's better ways of doing things than by this command, however, for some problems there are no substitues. A good example is that you want to take a look in the room that a player, for instance Adam, is standing in:

```
At adam look
```

You might want to take the precaution of going invisible before doing it so that most player-oriented output is forestalled.

2.5.2 Call a function in an object

This is a most particularly useful command. It enables you to call functions in an object with any parameters you like.

```
    }

    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.

2.4 Some mud commands

It might look strange that I've put this chapter so late in the text. The trouble really is that of the hen and the egg; It's impossible to understand and use the commands before understanding how the game really works. To understand how the game works, you need to experiment with the commands... Well, with any luck you were reading this manual before you started to experiment, so this chapter comes just in time for you to sit down and start writing some code.

First of all I'd like to ask you to bone up on your editing skills. Either learn how to use *ed*, which is the only available in-mud editor, or use an external editor together with ftp. I recommend emacs with ange-ftp for that purpose, makes network-based retrieving and saving files virtually invisible.

The following commands are best described in the game. Just type *help* <*command*> to get the full text. What I've written here is simply a short summation of their functionality.

```
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);
```

2.3.15 Directory handling

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]);
    }
```

2.3.16 Screen input/output

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

```
 * 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'.

2.3.13 Math functions

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.

```
/* 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 facilitates 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.

2.3.12 Type conversion

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 occasions 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 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");
```

2.3.11 Mapping functions

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,

and a string as arguments, 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
```

2.3.10 Array functions

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
     */
```

'@' 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 closely 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.

- * matches any number of any characters
- ? Matches any single character
- [xyz] Matches any characters in xyz
- [^xyz] Matches any characters not in xyz
- \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
```

2.3.7 Bit functions

Sometimes it's desirable 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;
```

```
}
```

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. Particularly 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.

```
 * 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))
```

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)
```

2.3.4 Alarms: Asynchronous function execution

Sometimes 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 existence 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 within 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.

NB! A 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 use that. Remember this since some efuns rely on a defined `this_player()` value. A lot of tricky bugs has been traced back to this particular problem in the past, so beware!

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, preferably two, as well as delays to single alarms.

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()
    {
```

```
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.

2.3.3 Object-inherent command handling

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.

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");
```

2.3.2.4 Object references to interactive objects

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");
```

```
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 validating 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("/w/Wizard/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 these 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.

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.

NB! Remember that *any* object you have cloned and distributed in the game will continue to exist unchanged even though you destroy the master object. New objects that you clone from the updated master object will have the new functionality, but the old ones will remain the way they were previously. This means that if you have discovered a real bummer of a bug in some kind of object that you have distributed to a lot of players, you'd need to create another specialized object to find them and replace them with the new version to actually get the bug removed from the game.

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.

2.3.2.1 Object references relative to the current 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.

'/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

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.

2.3.1.1 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:

- 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.
- 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.
- 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.
- 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 accessible 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.

```
            write("WOOOOPS! You did it!\n");
            break;

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

2.2.8 catch/throw: Error handling in 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 desirable 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 occurred 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");
```

2.2.9 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.
```

2.2.5.2 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 inaccessible for other objects.

2.2.5.3 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.

2.2.5.4 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.

2.2.5.5 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.

2.2.5.6 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`.

2.2.6 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;
```

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.

You can even add up more than one `::` instruction on top of another to reach higher levels. Each `::` only means 'back up one inheritance step, and look for the function there. If it's not there, look further up'. So, `::::::func()` simply means 'back up three inheritance steps and look for `func()` there, if not found, look even further up until you find it'.

If you have managed to confuse matters by having a function with the same name as an efun, the compiler needs a bit of help in knowing which function you actually mean. Now, it isn't exactly the brightest thing you can do to put yourself in this situation, but you will even find examples of it in the mudlib (in `/std/player/savevars_sec.c` to be specific, and I even *cough* seem to remember that it was I who wrote that part once upon a time), so I guess even the sun has spots... Anyway, to solve the problem, simply precede any such efun call with the instruction `efun::`, and the compiler will know what you mean.

```
void
deep_inventory(object ob)
{
    write("Dumping inventory of object '" + file_name(ob) + "'");
    dump_array(efun::deep_inventory(ob)); /* Call to real efun */
}

int
my_fun(string arg)
{
    object ob;

    if (objectp((ob = find_object(arg))))
        deep_inventory(ob);    /* Call to local version above */
}
```

2.2.3 Masking functions in runtime, part 1

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 in the shadow object that also exist in the shadowed object mask the original. Calls

2.2 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.

2.2.1 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.

2.2.1.1 Making object-internal function calls

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;
```

2.2.1.2 Making single object-external function calls

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>-><function>(<argument list>);
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);
```

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. The call will also fail if the calling object lacks the proper priviliges to load the target object.

2.2.1.3 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>-><function>(<argument list>);
e.g.
    /*
     * I want a mapping where the indices are the names of the players
```

```
#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.

Please consider that adding these statements reduces the legibility and thus also the quality and long-term maintainability of the code. Keeping them to a minimal is recommended.

1.3.4 What *not* to do with the preprocessor

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, since you include the uncompiled code into several different objects, 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.

When you include standard files, *always* use the <>-path notation. The reason isn't only that it becomes shorter and easier to distinguish 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.

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.

1.3.2 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 line break.

1.2.13.2 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

```
}

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

Sometimes 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;
}
```

1.2.13 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.

1.2.13.1 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).

```
        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 *definitely* isn't an improvement:

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

1.2.10 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

1.2.11 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.

1.2.11.1 The for statement

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

1.2.9.1 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;
```

1.2.9.2 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;
```

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 decrementation 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.

1.2.7.3 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 arithmetic) and all will be clear.

```
// 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 >
}
```

```
 */
<return type>
<function name>(<argument list>)
{
   <code expressions>
}

/*
 * 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 received 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);
}
```

1.2.5 Statements and Expressions

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

1.2.5.1 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");
```

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 preferable 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 turn out quite the way you wanted. It also makes the code easier to maintain in the long run, as modifiations of the existing code less easily introduces sudden undetected but incompatible changes.

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 fossils 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 value pair 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 facilities exist to make them easy and quick to use.

The difference between having interpreted and compiled code is that while the compiled code is quicker, the interpreted version is much easier 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.

1.1.3 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`').

1.1.4 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 object. 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.

1.1.5 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

```
(defun-block-intro     . +)
(class-open            . -)
(class-close           . 0)
(inline-open           . +)
(inline-close          . 0)
(ansi-funcdecl-cont    . +)
(knr-argdecl-intro     . +)
(knr-argdecl           . 0)
(topmost-intro         . 0)
(topmost-intro-cont    . 0)
(member-init-intro     . +)
(member-init-cont      . 0)
(inher-intro           . +)
(inher-cont            . c-lineup-multi-inher)
(block-open            . 0)
(block-close           . 0)
(brace-list-open       . 0)
(brace-list-close      . 0)
(brace-list-intro      . +)
(brace-list-entry      . 0)
(statement             . 0)
;; some people might prefer
;;(statement             . c-lineup-runin-statements)
(statement-cont        . +)
;; some people might prefer
;;(statement-cont        . c-lineup-math)
(statement-block-intro . +)
(statement-case-intro  . +)
(statement-case-open   . 0)
(substatement          . +)
(substatement-open     . 0)
(case-label            . 0)
(access-label          . -)
(label                 . 2)
(do-while-closure      . 0)
(else-clause           . 0)
(comment-intro         . c-lineup-comment)
(arglist-intro         . +)
(arglist-cont          . 0)
(arglist-cont-nonempty . c-lineup-arglist)
(arglist-close         . +)
(stream-op             . c-lineup-streamop)
(inclass               . +)
(cpp-macro             . -1000)
(friend                . 0)
(objc-method-intro     . -1000)
(objc-method-args-cont . c-lineup-ObjC-method-args)
(objc-method-call-cont . c-lineup-ObjC-method-call)
)))

;; emacs end
```

straight throughout the code. If we all agree on something, it's that wavy indention-levels is something not to be tolerated.

As it happens, using almost any of the freely available editors for writing C-code (I recommend emacs) will take care of this problem for you, and at the same time provide some rudimentary syntax-checking functionality.

4. Several arguments on a line separated by a comma or a semicolon have a space following the comma or semicolon. 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. Lump 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. While modern editors will do a lot of work here, it doesn't really hurt to reak long lines of code in proper places so that they don't wrap on their own beyond the end of a 80-width screen. Reading the code becomes a lot easier when you help out a bit.
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>
 *
 */
```

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 with using the all-inclusive male one.

iii - History of LPC

LPC is the interpreted mud language created by Lars Pensjö for his invention LPMUD, an interactive multi-user environment suited for many purposes, games not being the least of them. Since the first appearance in 1988 the language has changed dramatically.

Development was taken over by other people at Chalmers Datorförening, primarily Jakob Hallén, Lennart Augustsson and Anders Chrigström 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 conversely 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 intangible as a guild or sect that the players can become members of.

Within a domain there is a leader, a domain Liege. He is the local project leader 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.