LPC (Programming Language)

Subjects: Computer Science, Information Systems
Contributor: HandWiki Zhu

LPC (short for Lars Pensjö C) is an object-oriented programming language derived from C and developed originally by Lars Pensjö to facilitate MUD building on LPMuds. Though designed for game development, its flexibility has led to it being used for a variety of purposes, and to its evolution into the language Pike. LPC syntax places it in the family of C-like languages, with C and C++ its strongest influences.

Keywords: programming language ; lpc ; lpmuds

1. Basic Structure

Almost everything in LPC is an object. However, LPC does not precisely use the concept of a class (MudOS has something called a class, but it is really a struct). Instead, LPC objects are blueprint objects and **clones** of blueprint objects, in a prototype-based programming model. One can treat a blueprint object much as a class in other object-oriented languages.

Each object has variables (attributes) and functions (methods). The variables store the object's state; the functions are executable routines that can be called in the object. An object is uniquely defined by the name of the file in which it comes from, plus, if a clone, a numeric identifier. In a typical implementation, a clone of the file `/lib/weapon.c` which is the third clone created in the current run session will be `/lib/weapon#3`. The blueprint object corresponding to `/lib/weapon.c` is simply `/lib/weapon`. In a MUD game, common objects include rooms, weapons, armor, and non-player character (NPCs). Most mudlibs define inheritable objects for such common things. In the LPMud 2.4.5 mudlib, for example, the parent object for all rooms is `/lib/room`.

In LPC, it is not possible to syntactically declare class-only or instance-only methods and attributes; all functions and variables may be accessed identically in blueprints and in clones. For example, `"/lib/user"->number_of_users()` would call the `number_of_users()` function directly in the blueprint instance of `/lib/user.c`. (However, a controversial mechanism in many drivers called "shadowing" allows instance methods to be emulated to some extent by permitting one object to "overlay" its function set onto another, in effect interposing itself between external function calls and the shadowed object.) Direct external access to variables is also not supported; all interaction between objects is carried out through function calls (notwithstanding that data structures *pointed* to by variables are independent of objects and changes to them are visible in all objects referencing those data structures simultaneously, and that some drivers have privileged functions allowing a cumbersome form of external variable access that permits inspection but not mutation).

1.1. A Simple Room in a Typical Mudlib

Because LPC is generally used to code MUDs, 'rooms' are often created as objects that store information describing a particular scene along with exits that point to other room objects. This is the most common usage of LPC. Other uses that are not game related are possible.

The following example shows a very simple traditional room that leverages functions defined in the mudlib object `/lib/room`. However, not all mudlibs define or determine rooms in the same way, so this is not the only method of defining a room.

```
inherit "/lib/room"; void create() {    ::create(); set_short("a simple room"); set_long("A simple room in a simple building."); set_description("This is a simple room in a simple building. It is very nice."); add_exit("north", "/realms/descartes/north_room"); }
```

The first line tells the object to inherit functionality from the `/lib/room` object. This example assumes the `/lib/room` object defines functions for `::create()`, `set_short()`, `set_long()`, `set_description()`, and `add_exit()`.

This example contains a single function, `create()`. Most drivers call, or can be set to call, `create()` to allow an object to initialize itself with startup values; it is the standard constructor. In this case, the example calls functions that set up the basic room attributes in the inherited `/lib/room`. The functions called here are highly dependent on the mudlib in use, since the mudlib defines the actual room parent.

1.2. Data Types

Common data types

objectAny LPC object, including both blueprints and clones. Available in all known drivers. Has no literal form as such.

```
object obj = clone_object("/std/weapon");
```

intAn integer, generally 32-bit (with many drivers, compiling to 64-bit architecture will result in 64-bit ints, but stability when compiled to 64-bit architecture is not a common feature in LPMud drivers). Available in all known drivers. Literal form is the bare number.

```
int number = 23;
```

stringA variable-length character string. One does not need to do any form of memory management or bounds management; LPC handles that. Available in all known drivers. Literal form is the character string enclosed in double-quotes.

```
string text = "Hello, world!";
```

array, declared as <type> *An array of another data type. In most drivers, an array is defined using a type with the `*` modifier, as in `object *, string *, int *`. Typically, only one `*` can be applied to a declaration, so if one wanted an array of arrays, the declaration would be `mixed *`. LPC arrays are generally "rubberband" arrays that can have their length modified after allocation. Arrays are almost universally available, though the preferred declaration syntax differs in MudOS. Usual literal form is the list of values, separated by commas (optional after the last value), enclosed in `({` and `})` boundary markers.

```
int * numbers = ({ 1, 2, 3 }); string * words = ({ "foo", "bar", "baz" }); mixed *
stuff = ({ 1, "green", 48.2 });
```

mapping*See also: Comparison of programming languages (mapping)#LPC* A hash map (associative array) of keys to values. Supported by most drivers. Usual literal form is a `([` boundary marker, followed by zero or more key-value pairs with a colon between the key and value and a comma following the value (optional on the last key-value pair), and a `])` boundary marker.

```
mapping map = ([ "hello" : 1, "world" : 2, ]);
```

floatA floating-point numeric value, generally "single-precision" rather than "double-precision". Available in all known drivers.

```
float number = 3.1415;
```

mixedUsed to designate variables that may hold values of different types. Effectively, this disables compile-time type-checking on the variable. LPC has strong dynamic typing; one would work with mixed type variables by checking type information at runtime. Available in all known drivers.

```
mixed value = random(2) ? "hello" : 1;
```

Less common data types

statusMeant to hold a 1 or a 0 and thus represents true or false; equivalent in intention to a **boolean** type. In most drivers it is implemented as an int, and so actually can hold other values. It is deprecated in MudOS, largely because of this

inconsistency, and is not supported at all by Dworkin's Game Driver (DGD).

```
status flag = 1;
```

closureThe most common function-pointer data type, mainly known to be supported by the Amylaar LPMud and LDMud drivers. The basic literal syntax for obtaining a closure of a function is a hash mark, followed by a single quote, followed by the function name. Constructing lambda closures is much more complex.

```
closure func = #'some_local_function;
```

symbolUsed by drivers supporting lambda closure construction (mainly the Amylaar LPMud and LDMud drivers) for binding to variable namespace within lambda closures. Literal syntax is a single quote followed by the symbol name.

```
symbol var = 'x;
```

*quoted array*Also used by drivers supporting lambda closure construction, this is a special type used for designating arrays so that they are not interpreted as closure instructions. Literal form is a single quote followed by an array literal. Has no type declaration keyword, so variables intended to hold a quoted array would be declared *mixed*.

```
mixed quoted_array = '({ 1, 2, 3 });
```

<*type*> arrayThe form of array declaration preferred under MudOS. The `<type> *` form is usually also available.

```
int array numbers = ({ 1, 2, 3 });
```

class <*name*>Unique to MudOS; behaves like a C struct. It is not a class in any object-oriented sense of the word.

```
class example { int number; string name; }; class example instance = new(class example); instance->number = 23; instance->name = "Bob";
```

struct <*name*>Unique to LDMud 3.3 and up; behaves like a C struct.

```
struct example { int number; string name; }; struct example instance = (<example>); instance->number = 23; instance->name = "Bob";
```

functionThe MudOS function-pointer type, similar to **closure**, which is usually supported as an alias. Anonymous functions may be specified within `(: :)` boundaries, using `$1, $2,` etc. for arguments. LDMud also supports a variant of this syntax for its closure values.

```
function op = (: return sqrt($1 * $1 + $2 * $2);  :);
```

Pseudo-types and type modifiers

voidUsed as the declaration type for the return value of a function, specifies that the function will not return a value. Available in all known drivers.

varargsA type modifier for function declarations; specifies that the function's argument list is intended to be variable-length. Supported by most drivers. Some drivers support an extension to this behavior where a variable in the argument list of a varargs function can itself receive the varargs modifier, causing it to act as a "catch-all" variable for multiple arguments in the actual argument list, which it receives packed into an array.

privateA type modifier for object variable and function declarations; specifies that the affected entity should not be available in the namespace of any inheritors or, in the case of functions, be callable by other objects. Supported by most drivers.

staticA type modifier for object variable and function declarations. For variables, specifies that they should not be serialized. For functions, specifies that they should not be callable by other objects. Supported by most drivers, but often deprecated in favor of more sensibly designed type modifiers.

protectedA type modifier for function declarations; specifies that the function should not be callable by other objects, though it still appears in inheritors' function namespace, unlike the behavior of **private**. Supported by many drivers.

nosaveA type modifier for object variable declarations; specifies that the variable should not be serialized. Supported by many drivers; in such drivers, **static** for variables is generally deprecated in favor of **nosave**.

nomaskA type modifier for function declarations; specifies that it should not be permitted to override the function or otherwise obscure it in the function namespace. It is similar to the **final** type modifier in Java or PHP. Supported by

most drivers.

publicGenerally the default behavior, allowing unrestricted access to a variable or function, it can also usually be applied as a type modifier that prevents public access to the variable or function from being overridden later. Supported by most drivers.

virtualA type modifier for inherits; specifies that when a module occurs more than once in the inheritance tree, only one instance of its variables should be maintained. Supported by many drivers.

deprecatedA type modifier for object variable and functions declarations; specifies that the affected entity is deprecated and should not be used anymore. Any usage will cause a warning. Supported by LDMud 3.5.x.

Most drivers also support applying type modifiers to inherit statements, causing the inherited object to behave with respect to its inheritor as if the type modifier were applied to its functions and/or variables, as appropriate.

Where the term "object variable" is used above, this means a variable which is an element of an object (i.e. an attribute), as opposed to a local variable (exists only within a function or block) or a global variable (nonexistent in LPC — if someone speaks of a global variable in reference to LPC, they probably mean an object variable).

1.3. Passing Values

Primitive LPC types (int, string, status, float, etc.) are passed by value. Data structure types (object, array, mapping, class, struct) are passed by reference.

This feature can be powerful, but it can also lead to security holes. In most MUDs, the people building the world are generally less trusted than the staff running the game. If an object passes a mapping with sensitive values like access control information, an author of another object can modify that and thus increase their access rights. Mudlib developers and server administrators should thus be careful when passing complex types to lower access objects.

1.4. Function Types

LPC environments generally categorize functions into several major types according to how they are implemented:

lfunAn lfun, or "local function", is defined by a blueprint object. (Clones have the same function set as their blueprint.) They are written in LPC. Functions in a given object can call other functions within the same object using the syntax `function_name()`, while functions in other objects are usually called with the syntax `object->function_name()`. Overloaded lfuns defined in objects one is inheriting can be called with the syntax `::function_name()` or `object_name::function_name()`.

efunAn efun, or "external function", is defined by the driver. Efuns are written in C and compiled statically into the driver, so they generally run much faster than other function types, but are more difficult to write and lack flexibility. They are available in the namespace of all functions written in LPC, so, for example, the efun `this_player()` can be called with the syntax `this_player()`, or `efun::this_player()` if one needs to bypass an lfun or simul_efun.

simul_efun, sefunA simul_efun or sefun, "simulated external function", is written in LPC within the driver environment and placed in a special object whose functions mimic efuns for purposes of syntax. So an sefun `some_function()` is available as `some_function()` in the namespace of all LPC functions.

kfunDworkin's Game Driver (DGD) uses the term kfun, "kernel function", rather than efun. DGD kfuns are mostly identical with efuns in other implementations.

autoDGD does not precisely have simul_efuns, but rather has an "auto object" that acts as if it is automatically inherited by all other objects. This partially mimics the behavior of simul_efuns in other implementations.

1.5. Master Object

LPC implementations generally have a "master object", which is a specific LPC object that is loaded first by the LPC driver and which essentially controls what will happen past that point. The master object will typically tell the driver where the simul_efun object is, preload any objects which need to be present at startup, define what functions will be called when an object is loaded, and otherwise configure the driver's operation. The driver will refer back to the master object when it needs to interact with a central point of reference for the running environment, such as for accepting network connections, handling errors, and validating attempts to perform privileged operations.