NannyMUD LPC

Mats H. Carlberg

March 1998

Contents

1 Introduction

Welcome to the textbook on NannyMUD's LPC. As the name implies, this deals with LPC in the special flavour used by NannyMUD. It is possible that some of what is written here also pertains to other MUDs, but don't trust that.

The reader is assumed to have some familiarity with other programming languages, and programming in general. This is not a document teaching programming from scratch, nor will it deal with algorithms etc. There are many excellent book on those topics that you can read instead.

This text was written to mimic the book 'The C programming Language' by B.W. Kernighan and D.M. Ritchie. I thought there was a need for a short and compact text, concentrating on the LPC language, less clogged by code examples than most of other documentation on the subject I have seen. There are many variants of LPC available, depending on the drivers one use: DGD, MUDOS, Amylaar, etc. This document concerns itself with the NannyMUD version of LPC; to cover all of the various dialects would produce a gross tome indeed. The driver available when this was first written was 1.15.3; it is possible that future drivers will force a revision of the texts within this document.

Much of the text is based on the documentation available on-line in NannyMUD. Some phrases have been copied from there, when found suitable. Other has been rewritten, reformed, expanded etc. Some formulations are very close to those used by Kernighan and Ritchie; their book has been a source of inspiration.

This document was written using the GNU Emacs editor and typeset using the LaTeX typesetting system by L. Lamport, which in turn uses the TeX system by D. Knuth.

Linköping 1998,
Mats H. Carlberg
(Brom@NannyMUD)

The following people (in no special order) contributed to this work with ideas, corrections, religious debates about lexical details, tea, good company, and proofreading of the text:

Catarina Carlberg	Banshee@NannyMUD
Joakim Björklund	Taren@NannyMUD
Jones Desougi	Gwendolyn@NannyMUD
Mats Person	Mats@NannyMUD
Peter Skov	Qqqq@NannyMUD

2 Fundamentals

2.1 What is LPC?

LPC is an acronym for 'Lars Pensjö C', and was in its original form created by Lars Pensjö in 1989. LPC is an *object-oriented (OO)*, interpreted language, and the interpreter program is known as the *driver*. The version number of the driver is also the version number of the LPC. NannyMUD started off with the 2.4.5 driver on the spring of 1990.

2.2 LPC object and times

The definition of an LPC object is a text-file. When the driver detects that the object is needed, the driver first checks if the object is already in the memory. If it is not, the file is read and parsed into an internal representation. This first step is called *loading* the object, and the moment in time when it happens is known as *load-time*. After loading, a function named 'reset' is called in the object, with the argument zero.

The object stays loaded in memory until explicitly destructed. When there is a need to execute code in the 'loaded' object, the internal representation is interpreted by the driver. This happens in *run-time*.

The loaded object can then, after being loaded, be copied in a process called *cloning*. The original object, i.e. the loaded one, is known as *the master object*. Cloned objects are called *clones*.

2.3 The value of truth

In LPC, there is no such thing as a special truth value, nor any special false value. The distinction is simply between zero, and everything else. This has several implications which will become clear further on.

2.4 More Reading

Inside NannyMUD, there is detailed and voluminous documentation available to the coders through the use of the 'man' command. The most up-to-date documentation is, naturally, the source code of the objects. At the time of writing this document, both the man-pages and the source code are available over WWW. Navigate down from http://www.lysator.liu.se/nanny/ to access it.

3 Types, operators and expressions

3.1 Identifier Names

Valid identifier names are formed from a combination of the characters a-z, A-Z, underscore, and the numbers 0-9. They must start with a letter, or underscore. Case is significant, 'x' and 'X' are two different variable names. The keywords of the LPC cannot, naturally, be used as variable names. A list of such reserved keywords can be found in appendix A.

3.2 Type Checking

Global variables that are explicitly initialised when declared are typechecked at load-time. Apart from that, there is normally no global typechecking done in LPC, neither at load-time, nor at run-time. There are, however, two #pragmas that can be declared to get global type-checking at load-time: *#pragma strict_types* and *#pragma save_types*.

The first turns on typechecking at load-time, while the latter saves the type information so it can be used by inheriting objects. Unless those are used, the types given in the source code file is just a kind of documentation.

Local type-checking can be forced by declaring the types of functions; this is covered in section 5.

3.3 Data Types and Modifiers

All variables are initialised to zero (0). Variables, functions and expressions in LPC can have one of the types *int*, *status*, *float* (but see below), *string*, *object*, *mapping*, and *mixed*. There is one extra type available for functions, *void*. That, and type-checking rules for functions, are covered in section 5.

It should perhaps be emphasised that LPC does not, despite the existence of the type 'float', use floating point numbers. The various data types somewhat more in detail:

- int
  This is a signed integer. Examples of integers are 0, 4711, -5723, 0x7ffffff and 'c'.
- status
  This should really be a boolean, and accept only the values 0 and 1, but it is a full int.
- float
  This is just another name for 'int', and a bad one at that. Don't fool yourself, or others; do not use this.
- string
  This is a string, and not (as in C) a pointer to a string.
- object
  This is a pointer to an object. If the object is destructed, the pointer will be zero.
- mapping
  This is a mapping, the LPC word for an associative array. More will be said about mappings in section 6.3.
- mixed
  This is a way of telling the driver that you don't know what type this variable will have. A variable of this type can have any of the types above, and it can be explicitly casted to a specific type.

It is also possible to have arrays of variables of any type. Arrays and mappings are treated in the section 6. Several modifiers can be applied to the variables: *static*, *private*, *public*, and *nomask*.

- static
  A variable declared as static will not be saved nor restored using the efuns[1] save_object()[2] and restore_object()[3].

- private
  This can be given for both functions and variables. Functions that are private in an object, A, cannot be called through call_other[4] from any object, not even by the defining object itself. Also, they are not accessible to any object that inherits A.

- public
  A function defined as public will always be accessible from other objects, even if private inheritance is used. See section 7.1 for details on inheritance.

- nomask
  A symbol defined as nomask cannot be redefined by inheritance. It can still be used and accessed as usual.

3.4 Constants

An integer constant like 4711, is an integer. Likewise is -12, 'c' (with the value 99), '\077' and 0xff (hexadecimal representation of 255). Integer constants overflowing the range are set to the max range. The type status behaves exactly the same way, since it is in really an integer.

String constants are a set of characters surrounded by double quotes, ". Examples of string constants are "hello world!", "\077", and "\\ \"a\n". Note that a character in a string can be given using octal representation, but not using hexadecimal.

Array constants are comma-separated lists within the delimiters ({ and }). For example, ({ 4, 47, 471, 4711 }) is an array of integers, ({ "a", 1 }) is a mixed array with an int and a string, and ({ ({ 13 , 17 }), ({ 23 }) }) is an array of arrays of integers.

The mapping constants are comma-separated lists of key-value pairs surrounded by ([and]). Examples of mapping constants are ([1:12, 5:"abc"]) and ([({ "h" }):({ 1, "foo" }), ([5:77]):"foo"]).

There are basically no object constants, apart from the value 0.

3.5 Declarations

All variables must be declared before they are used. A declaration contains a type, followed by a comma-separated list of variable names. For example,

```
int i, j, k, l;
int a;
```

[1] See section 5.1

[2] 'save_object()' is a function provided by the driver to dump a representation of an objects variable and their values to file.

[3] 'restore_object()' reads an objects variables and values from file.

[4] 'call_other' is a function provided by the driver to call functions in a specified object. The specified object can, naturally, be the calling object itself. That might seem a waste, but consider 'shadows' (see 8.1).

declares the variables a, i, j, k, and l, as being integers.

Global variables (i.e. the variables declared outside any function) can be declared in any place, as long as the declaration precedes their first use. Global variables can also be initialised when declared (and are then type-checked):

```
string tmp_s = "foo faa fuu";
```

declares a string variable tmp_s, and initialises it to have the value "foo faa fuu".

3.6 Arithmetic Operators

There are several arithmetic operators available in LPC: +, -, *, / and %.

- The + operator
  Unary +, as in +12, is allowed (on integers) but has no effect. The binary use, *expression1 + expression2*, works on integers, strings, arrays and mappings, as well on the combination string/integer.

  For integers, the result is the usual arithmetic sum. For strings, the result is the concatenation of the two strings. Adding two arrays, the result is an array containing the elements from both arrays. Addition of mappings work similarly to the addition of arrays. Addition of an integer and string converts the integer to a string and concatenates the two strings.

- The - operator
  Used as an unary operator, *-expr*, the operator changes the sign on the value of the expression. In this case, it works only on integers. Used as a binary operator, *expr1 - expr2*[5] it works on integers and arrays. For integers the result is the common arithmetic difference, and for arrays it is the elements of 'expr1', with all elements of 'expr2' excluded.

- The * operator
  The * operator has only a binary use: *expr1 * expr2*. It works on integer values only, and then gives the usual arithmetic product.

- The / operator
  The / operator has only a binary use: *expr1 / expr2*. It works on integer values only, and then gives the integer division.

- The % operator
  The % operator has only a binary use: *expr1 % expr2*. It works on integer values only, and then gives the remainder of the integer division.

3.7 Relational and Logical Operators

The relational operators are >, >=, <=, <, == and != .

[5] Strictly speaking, it is the value of expression1, etc., that is used. Pointing this out every time makes the text rather cumbersome, so we adopt the slightly sloppy habit of letting this be understood by the context.

- == and !=
  Those operators work when comparing all datatypes. For integers and strings, equality means that they have the same value. For objects, equality occurs when two expressions yield a pointer to the same object. For mappings and arrays, equality occurs when two expressions yields the same memory (see also sections 6.2 and 6.3).

- <, <=, >= and >
  The operators < and > works on both integers and strings. Integers are compared as usual, while strings are compared using the ASCII collational sequence. For example, "a" is less than "b", and so is "aa".

- ||
  This is the logical 'or' operation. The result of *expr1 || expr2* is true if 'expr1' is true (in which case 'expr2' is *not* evaluated), or if 'expr1' is false and 'expr2' is true.

  Remember that anything that isn't zero is considered true. Thus, an existing object, an empty array or mapping, and an empty string are all considered true.

- &&
  This is the logical 'and' operation. The combinded expression *expr1 && expr2* is true if both expressions are true. 'expr2' is *not* evaluated if 'expr1' is false.

- !
  This is the logical 'not' operator. It turns any true value into zero, and zero into one.

- ~
  The operator ~ is the boolean not operator (ones complement), an unary operator. It works solely on integers.

3.8 Increment and Decrement Operators

The increment, ++, and decrement, --, operators works on variables with type integer.

- ++var
  This increments the value of variable 'var', and returns the *new* value to the expression.

- var++
  This increments the value of variable 'var', and returns the *old* value to the expression.

- --var
  This decrements the value of variable 'var', and returns the *new* value to the expression.

- var--
  This decrements the value of variable 'var', and returns the *old* value to the expression.

3.9 Bitwise Operators

There bitwise operators are |, ^, &, << and >>.

- |
  This is the bitwise 'or' operation. It works on integers only.
- ^
  This is the bitwise 'xor' operation. It works on integers only.
- &
  This is the bitwise and operation. It works on integers, and has been extended to work on arrays. In the latter case, the result is an array that holds the elements that occur in both arrays, i.e. the intersection.
- <<
  This is the left shift operator; *expr1* << *expr2* shifts 'expr1' left 'expr2' bits.
- >>
  This is the rights shift operator; *expr1* >> *expr2* shifts 'expr1' right 'expr2' bits.

3.10 Assignment Operators and Expressions

The assignment of a value to a variable is usually done through the construction *var* = *expr*. The value of 'expr' is assigned to the variable 'var'. The value of the whole is the new value of the variable 'var'.

The expression

```
i = i + 4;
```

can be written in a shorter form:

```
i += 4;
```

where the operator += is known as an *assignment operator*. There are several assignments operators available in LPC: +=, -=, *=, /=, %=, &=, ^=, |=, <<= and >>=.

If 'expr1' and 'expr2' are expressions, and 'op' one of the above operators,

```
expr1 op= expr2;
```

is equivalent to

```
expr1 = (expr1) op (expr2);
```

but 'expr1' is evaluated only once.

3.11 Other Operators

There are a few operators available that do not fit into any of the above categories.

- ,
  The comma operator, *expr1* , *expr2*, computes the value of 'expr1', throws away that value and then computes 'expr2', which value is the value of the total expression.

- [ ]
  The index operator, [ ], works on arrays, mapping and strings. For the use of index on strings, arrays and mappings, see the discussion in the section 6. Index starts at zero, always. Negative index counts from the end of the string/array. You can use ranges as index.

- ..
  This is the range operator, and it can only be used within an index operator. It can be used to pick out a certain range from an array, or a sub-string from a string. Both the start and the stop position must be given, both must be zero or positive, and stop must be larger than or equal to the start position.

- -> This operator is called 'call-other'. It's usage is *expr1 -> name(...)*. 'expr1' can either be an object, or a string. In the latter case, an object is created by loading the file named by 'expr1'. Then the function 'name' in that object is called. It is then a 'call function in another object', or 'call-other' for short. It does exactly the same work as the efun[6] 'call_other()'[7]

- (type)
  If an expression is of the type mixed, it might be possible to convert it to another type using the cast operator, *(type)expr*. This only work if the type of the expression fits into the type of the cast.

- ::
  This operator is used to call functions in an inherited object. It can be used in two forms: *::function()* and *filename::function()*. The latter specify exactly in what inherited object the function is called, which can be of interest in a situation where several objects are inherited.

3.12 Conditional Expressions

The statements

```
if (expr1)
  expr2;
else
  expr3;
```

can be written as an expression using the ternary operator *:?* : *expr1 ? expr2 : expr3*. This is truly an expression and can thus be used wherever an expression can be used. It makes for stream-lined code, but can also reduce the readability of your code drastically.

3.13 Precedence and Order of Evaluation

Here is the list of operators in LPC, listed in descending precedence, from left to right and top to bottom (i.e. :: has higher precedence than ->, and [ ] has higher precedence than --).

[6] See section 5.1.

[7] In fact, it is exactly the same thing, written in two different ways.

::	->	[ ]
-- (post-decrement)	++ (post-increment)	~
!	- (unary minus)	-- (pre-decrement)
++ (pre-increment)	(type) (type cast)	/
%	*	-
+	>>	<<
<=	<	>=
>	!=	==
&	^	|
&&	||	?:
/=	%=	*=
>>=	<<=	^=
|=	&=	-=
+=	=	,
..		

The order of evaluation of expressions are basically from left to right. The precedence of operators can of course change this. Unlike many other programming languages, LPC has the property that in the case of two expressions having the same precedence, they are evaluated left to right.

Thus, for example, there is no ambiguity of what the value is of the subscript in

```
a[i] = i++;
```

as it will always be the old value.

3.14 The time cost of everything

The MUD process is single-threaded. In order to prevent any piece of LPC-code to be able to hang the MUD, every operation performed when interpreting the code has been associated with a cost, called 'nodes'. The sum of evaluated nodes is not allowed to pass a certain limit, determined when the driver is compiled, known as 'max eval cost'. It is good to know the limit is there; the exact value is not that important.

Basically, the association of the cost with the operation is arbitrary in that it is not based on real life time; rather it is based on the driver hackers (sourcerers) whims.

4 Flow Control

The flow-control statements of LPC determine the order in which the code will be executed. There are several different ways of doing this. Some differ just in how the code look, while other have an impact on performance.

4.1 Statements and Blocks

A *statement* is an expression followed by a semi-colon, ';'. Curly braces, '{' and '}' are used to group statements into *blocks*, which are also known as *compound statements*. Syntactically, a block is equivalent to a single statement.

4.2 Conditional execution: If-Else

The if-else statement is used to allow one expression to determine what statements should be executed. It has the form

```
if (expr)
  statement1
else
  statement2
```

with the 'else' part being optional. If the value of 'expr' is true (i.e. non-zero), statement1 will be executed, else statement2 will be executed (if the else-branch exists).

The else-branch, when it exists, is associated with the latest else-less if. If that is not what you want, you need to use '{' and '}' to force the correct association.

4.3 Conditional execution: Else-If

A rather common construction is to let the optional 'else' part of an if-else statement be another if-else statement. One then arrives at something like

```
if (expr1)
  statement1
else if (expr2)
  statement2
else if ( ... )
  ...
else
  statementN
```

This chain is terminated as soon as an expression is true and the corresponding statement has been executed. This is a more general solution than the 'switch' statement discussed further on.

4.4 Conditional execution: Switch

In LPC, a switch-statement is a multi-branch that tests whether an expression matches one of a set of *constant* integers or strings.

```
switch(expr)
{
  case constant1 : statement1
  case constant2 : statement2
  ...
  case constantN : statementN
  default: statementD
}
```

The statement after the constant matching 'expr' is executed. If there is no match, the statement following the 'default' label is executed. The default part is optional.

Since the matching only determines the branching, all statements following the chosen branch at 'case constantI' will be executed, unless 'statementI' contains a 'break' statement. In a function, a 'return' statement will work nicely, too.

The 'constantI' can use the range operator to allow for a whole range of values. For example:

```
switch(random(10))
{
  case 0 .. 8 :
    return 1;
  default :
    return 0;
}
```

4.5 Loops: While

The while loop is the basic loop in LPC. It is simply expressed as

```
while (expr)
  statement
```

where 'expr' is evaluated, and if found true 'statement' is executed. Then 'expr' is evaluated again, etc., until it yields zero, at which point the program after 'statement' is executed.

4.6 Loops: For

The for loop

```
for (expr1; expr2; expr3)
  statement
```

is essentially equal to

```
expr1;
while (expr2)
{
  statement
  expr3;
}
```

All three of 'expr1', 'expr2' and 'expr3' are optional, but if 'expr2' is skipped, it is assumed to be true all of the time. The loop then goes on forever[8] unless terminated by other means.

[8] 'Forever' is not really true, 'until max eval cost' would be a better description. That feels like eternity anyway.

4.7 Loops: Do-While

Both the while and for loops test the expression at the top of the loop. The do-while loop on the other hand, does the test at the end of the loop:

```
do
  statement
while (expr);
```

In LPC, this is actually faster[9] than while and for loops.

4.8 Loops: Foreach

The foreach loop is the fastest of them all. It works on an array:

```
foreach(var, array)
  statement
```

The variable 'var' is set to each of the elements of 'array' in turn, and 'statement' is executed.

4.9 Break and Continue

There are two statements that can be used to alter the flow of a loop: break and continue. The continue statement makes the loop start over again at the top, while break exits the innermost enclosing loop (or switch) immediately.

5 Functions and Program Structure

LPC functions work in the same manner as functions in other languages; they make it possible to split a problem into many small parts, write code once, use others code and help make the code readable, etc.

5.1 Efuns, Sfuns and Lfuns

There are several classes of functions available to an LPC object:

- efuns
  Efuns are *external functions*, i.e they are defined outside any object, i.e. they are provided by the driver. They can be called by all objects, but the driver might return different things depending on what object made the call.
- sfuns
  Sfuns are *simulated efuns*. This means that the function behaves as an efun, i.e. is available to all objects, but that it is really implemented through some LPC code.
- lfuns
  Lfuns are *local functions*, i.e. they are defined by an LPC object. They will be discussed in detail in the following sections.

[9] At least on the driver used when this was written, the infamous 1.15.3.

5.2 Functions without type

Functions can be declared without specifying a type, unless the #pragma strict_types is in effect, in which case all functions must have types. Declared without types, the function arguments can be declared without types, too, and the function will accept that the actual number of arguments passed differs from the number of arguments declared. If called with too many arguments, the extras are ignored. If called with too few, the missing ones default to zero.

Functions declared without types can be called from any other object, they can be called from inheriting objects as well, and they can be overloaded[10].

5.3 Function types

A function can have the same types as a variable, and then one more: *void*. A function of type void cannot return any value; doing so is considered an error at load-time.

If the type of the function is specified, the type of the arguments must also be given. If the types of the arguments are specified, the type-checking usually given by #pragma strict_types will be performed on the statements of the function. Also, the number of arguments passed to the function must be the same as the number declared.

There is a set of type modifiers that can be used on the functions type: static, varargs, public, private

- static
  Functions declared with the type modifier 'static' cannot be called from other objects.
- varargs
  A function declared with this type modifier can be called with a varying number of arguments.
- private
  Private functions cannot be called from any other object, nor from any object inheriting the object where the function is defined.
- public
  A function defined as public will always be accessible from other objects.
- nomask
  A symbol defined as 'nomask' cannot be redefined by inheritance, nor by the mechanism of 'shadowing'[11].

5.4 Function Prototyping

When using the #pragma strict_types, functions must be declared before they are used. This will in many cases force functions to be declared using prototypes. This means that the functions type, as well as what parameters it takes and their type, is specified, but the body of the function (with a repetition of the type etc.) appears further on in the file.

[10] See section 7.2.
[11] See section 8.1 on shadows.

5.5 Functions Returning Strings, Arrays or Mappings

A function can be exited at any point by adding a *return expr;* statement at the desired point of the code. The value of 'expr' is then the value returned from the function to the calling code.

A function returning a string, mapping or an array does so by returning a reference to it, rather than a copy. This can be used to keep shared strings/arrays/mappings, thus saving some memory. If you'd rather want a copy, you will have to force it. This is easiest done by returning the sum of the string/array/mapping and an empty string/array/mapping.

5.6 Calling Non-Existant Functions

In LPC, calling a non-existing function in another object returns a zero. This is a feature. On the other hand, calling a non-existing function locally (in the same object) gives a run-time error. This is, also, a feature.

5.7 Scope Rules

The scope of functions and global variables is from their point of definition to the end of the file. The scope of formal parameters (those declared as part of a function declaration) is the whole of the function.

Local variables can be declared at the beginning of any block, and their scope is from their point of declaration to the end of the function, not just the end of the block where they were declared. This can be considered an unwanted behaviour. Local variables hide any global variables with the same name.

5.8 Block Structure

Functions cannot be declared within functions in LPC, but within the functions, blocks can be declared within blocks. Each block can begin by declaring variables, but local variables can only be declared *once*.

5.9 Recursion

Functions in LPC can be called recursively. The driver will allow a certain depth of the recursion before it stops the code. Circular recursive calls will also be detected.

5.10 The Preprocessor

The LPC preprocessor works like C's, almost. Somewhat more details:

- #pragma strict_types
  This turns on typechecking for the whole file (at load-time).
- #pragma save_types
  This makes the driver save type information which then can be used objects inheriting this object.

- #define name token-string
  Replace subsequent instances of 'name' with token-string.

- #define name(argument [, argument] ...) token-string
  This is a definition of a macro. Note that there can be no space between 'name' and the '('. Subsequent instances of 'name', followed by a list of arguments within parentheses, are replaced by token-string, where each occurrence of an argument in the token-string is replaced by the corresponding token in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded token-string unchanged. After the entire token-string has been expanded, the preprocessor re-starts its scan for names to expand at the beginning of the newly created token-string.

- #echo
  This creates an entry in the MUD's debug-log.

- #undef name
  Remove any definition for the symbol 'name'. No additional tokens are permitted on the directive line after name.

- #include "filename"

- #include <filename>
  Read in the contents of 'filename' at this location. This data is processed as if it were part of the current file. When the <filename> notation is used, 'filename' is only searched for in the /include[12]

- #line integer-constant "filename"
  Generate line control information for the next pass of the compiler. 'integer-constant' is interpreted as the line number of the next line and 'filename' is interpreted as the file from where it comes. If 'filename' is not given, the current filename is unchanged. No additional tokens are permitted on the directive line after the optional filename.

- #if constant-expression
  Subsequent lines up to the matching #else, #elif, or #endif directive, appear in the output only if 'constant-expression' yields a nonzero value. All binary non-assignment LPC operators, including '&&', '||', and ',', are legal in 'constant-expression'. The '?:' operator, and the unary '-', '!', and '~' operators, are also legal in 'constant-expression'.

  The precedence of these operators is the same as that for LPC. In addition, the unary operator 'defined', can be used in constant-expression in these two forms: 'defined (name)' or 'defined name'. This allows the effect of #ifdef and #ifndef directives (described below) in the #if directive. Only these operators, integer constants, and names that are known by the preprocessor should be used within 'constant-expression'. In particular, the sizeof operator is not available.

[12] Or rather, in a list of places defined by the mudlib. This list is usually determined by /obj/master.c.

- #ifdef name
  Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if name has been defined with a #define directive, and in the absence of an intervening #undef directive. Additional tokens after name on the directive line will be silently ignored.

- #ifndef name
  Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if 'name' has not been defined, or if its definition has been removed with an #undef directive. No additional tokens are permitted on the directive line after name.

- #elif constant-expression
  Any number of #elif directives may appear between an #if, #ifdef, or #ifndef directive and a matching #else or #endif directive. The lines following the #elif directive appear in the output only if all of the following conditions hold:

  – The 'constant-expression' in the preceding #if directive evaluated to zero, the name in the preceding #ifdef is not defined, or the name in the preceding #ifndef directive was defined.
  – The 'constant-expression' in all intervening #elif directives evaluated to zero.
  – The current 'constant-expression' evaluates to non-zero.

  If the 'constant-expression' evaluates to non-zero subsequent #elif and #else directives are ignored up to the matching #endif. Any 'constant-expression' allowed in an #if directive is allowed in an #elif directive.

- #else
  This inverts the sense of the conditional directive otherwise in effect. If the preceding conditional would indicate that lines are to be included, then lines between the #else and the matching #endif are ignored. If the preceding conditional indicates that lines would be ignored, subsequent lines are included in the output. Conditional directives and corresponding #else directives can be nested.

- #endif
  End a section of lines begun by one of the conditional directives #if, #ifdef, or #ifndef. Each such directive must have a matching #endif.

There are some things that works in the standard C preprocessor but that cannot be used in the LPC preprocessor:

- Concatenation of actual arguments during macro expansion using the '##' construction does not work.

- Actual arguments cannot be expanded to quoted strings using the '#' construction.

- A macro's body cannot contain the macro itself.

5.11 Comments in LPC

LPC allows comments over whole blocks by enclosing it in '/*' and '*/', just like in C. It also allows for comments to the end of line using the C++-style '//'.

6 Strings, Arrays and Mappings

6.1 Strings

LPC has real strings, not arrays of characters. Strings are shared, i.e. when a string is created it is stored in a table and replaced with a reference. This saves memory.

The index operator can be used on strings. The returned value of *str[i]* is the character (i.e., an integer) at position *i* in the string 'str'. For example, '"abcd"[-1]' gives 100, the character value of 'd'. Note that this is *not* the *string* "d".

The index operator can be used to set single elements in the string:

```
string str;

str = "abcdef";
str[1] = 'C'; // str is now "aCcdef"
```

but not to set a sub-string of length 1, i.e.

```
string str;

str = "abcdef";
str[1] = "C";
```

does not work; it will in fact trigger a load-time error. Note the difference between the character 'C' and the string "C".

Giving the index operator a range will yield a string:

```
string str1, str2, str3;

str1 = "abcdef";
str2 = str1[1..3]; // str2 is now "bcd".
str3 = str1[2..2]; // str3 is now "c".
```

In this way, the index operator *can* be used to set parts of a string:

```
string s;

s = "abcdef";
s[1..2] = "BC";   // s is now "aBCdef".
s[1..2] = "B--C"; // s is now "aB--Cdef".
```

6.2 Arrays

LPC allows you to declare arrays of any type. This is usually done by prepending a '*' before the variable name, as in 'int *many' (which declares an array of integers named 'many'). If you use the #pragma strict_types, you *must* declare them that way. Before you can use the array, you have to initialise it to a valid array. Remember that LPC initialises all variables to zero (0), which is not a valid array.

You can create an array by using the allocate efun:

```
int *arr;
arr = allocate(12);
```

which declares an array of ints 'arr' and initialises it to consist of 12 integers, numbered 0 to 11, that are zero. The use of 'allocate' is not needed, the above can as well be written

```
int *arr;
arr = ({ 0,0,0,0,0,0,0,0,0,0,0,0 });
```

Arrays change size dynamically. For example, adding two arrays with three elements gives a single array with 6 elements. The allocation of the needed memory and the copying of data is handled by the driver.

You can, naturally, have multi-dimensional arrays. Those are handled as arrays of arrays, and thus there is no real type for them. You then have to resort to using the type 'mixed'.

The index operator can be used on arrays to get a single element, a range of elements, to set a single element, and to set a range of elements. For example:

```
mixed a, b;

a = ({ 0,1,2,3,4 });
b = a[2];   // b is now 2.
b = a[2..3] // b is now ({ 2,3 }).

a[5] = 9;                // a is now ({ 0,1,2,3,4,9 }).
a[0..0] = ({ "a","b" }); // a is now ({ "a","b",1,2,3,4,9 }).
a[0..1] = ({ "c","d" }); // a is now ({ "c","d",1,2,3,4,9 }).
a[0..3] = ({ 99 });      // a is now ({ 99,3,4,9 }).
a[0] = ({ 0,1 });        // a is now ({ ({ 0,1 }), 3,4,5 }).

a = ({});
a[0..0] = ({ 1 }); // a is now ({ 1 });

a = ({});
a[0] = 0; // This gives an 'index out of bounds' error.
```

6.3 Mappings

A mapping is LPC's associative array. You can think of it as an array that accepts any type of value as an index, or as a list of index-data pairs. Using an index for which there is no data, returns a zero. New entries are simply added by

```
map[index]=value;
```

or by

```
map += ([ index:value ]);
```

The former over-writes any previous entry on that index, the latter does not. Using the latter, you can force several entries with the same index but different values. If so, what exact value is returned by 'map[index]' cannot be predicted[13]. Also of interest is that the former construction just extends the mapping, while the latter creates a *new copy* of the mapping.

The index operator does not accept ranges when operating on mappings.

Mappings are always sorted (on the indices, but the sort order is not one humans can interpret, with one glorious exception: when the index are integers, the mapping is sorted in ascending order.

6.4 Strings, Arrays and Mappings as Function Arguments

For all practical purposes, strings,arrays and mappings as function arguments are passed by value. In reality, they are passed by reference, and the copying is delayed until a change is done to the string/array/mapping.

7 Inheritance and Overloading

7.1 Inheritance

LPC supports inheritance. Inheritance is a way of making the variables and functions of one object available to another without copying the actual code. The syntax for inheriting an object is *inherit "filename";* where 'filename' is the filename of the object one wish to inherit. This statement must come before any variables and functions are defined. LPC allows the inheritance of multiple objects.

An object can, however, protect function and variables from access through inheritance by declaring them with the type modifier 'private'. The inheritance itself can be modified by the modifiers 'private' and 'static'. Only the 'static' modifier has any effects, and that is to mark the inherited variables as static.

7.2 Overloading

All inherited functions can be overloaded, or redefined, by the inheriting object. Redefining a function declared as having the type modifier 'nomask' triggers a load-time error.

The overloaded functions can still be accessed by prepending the function name by '::', *::function();*. If you inherit several files that defines the same function, this will call 'function' in the last inherited object. By prepending the '::' by the filename, exactly which function is called can be controlled: *filename::function*.

As an example, consider three files, 'file1', 'file2' and 'file3', all of which defines the function 'foo()', which returns 1, 2 and 3 respectively:

[13] It will not change while the driver runs, but differ between different runs. Shutting down the driver and restarting it is known as a 'reboot'.

```
inherit "path1/file1";
inherit "path2/file2";
inherit "path3/file3";

foo()
{
  int i;

  i  = file1::foo(); // Call function foo() in first object.
  i += file2::foo(); // Call function foo() in second object.
  i += file3::foo(); // Call function foo() in third object.
                     // This could as well have been written as
                     // i += ::foo();
                     // since 'file3' is the last inherited.
  return i;
}
```

would then return 6, the sum.

8 Miscelleanous Features

8.1 Shadows

Shadows are used to reroute the calls to lfuns through another object. In a sense, the functions in the shadowed object are overloaded by those in the shadowing object - for external function call (i.e. call_other, ->). The shadowee still has access to the original functions by internal calls.

This can be used to modify the returned value and the sideeffects of a function without having to re-define it, especially for objects that are not under your own control (i.e. you didn't write their definition files).

This means also that when designing an object, the coder has to consider how shadows can be used on and affect the behaviour of the object. It might be a good idea to block the possibility to shadow certain functions, or perhaps make the object impossible to shadow at all.

This involves a judgement call, as denying shadows increases security but decreases flexibility. For most objects, flexibility is more important than tight security.

8.2 Virtual Compiling

Virtual compiling allows for objects with distinctive filenames, without any corresponding file defining them. This might sound a little odd, but it has its uses.

When the driver is instructed to load a file, it first checks if it is already loaded. If not it looks in the specified directory. If there is no file there that can be used, it looks for a file named vcompile.c in the directory (and in the parent directories!). If found, that object is loaded and the function compile_virtual in it is called with the filename of the wanted object as argument. This function should clone an object and return it.

If anything else than an object is returned, the common error handling mechanism for a non-existing definition file is used. But if an object is returned, the driver now changes file name on the clone to the filename given as argument to compile_virtual, and the reset function in the object is called with the argument -1.

This can then be used to create large areas based on a database or some fancy algorithms instead of writing a large number of files defining every single location.

A The Keywords of LPC

Here is a list of the reserved keywords in LPC. This list was taken from the 1.15.3 driver; other drivers might reserve more, or less, keywords.

_acl_access	_acl_debug	_acl_get
_acl_modify	_acl_num2str	_acl_put
_acl_query_access	_acl_str2num	_cache_stats
_chmod	_destruct	_disconnect
_driver_stat	_dump_file_descriptors	_file_stat
_find_shortest_path	_fuzzymatch	_get_clone_by_number
_get_spec_obj	_isclone	_lock
_m_delete	_next_clone	_object_cpu
_object_memory	_object_stat	_query_action
_query_ed	_rusage	_set_prompt
_syslog	_wiz_list_info	add_action
add_verb	add_worth	add_verb
all_inventory	allocate	assoc
atoi	break	break_point
call_other	call_out	call_out_info
capitalize	case	cat
catch	clear_bit	clone_object
command	continue	create_wizard
creator	crypt	ctime
debug_info	deep_inventory	default
destruct	disable_commands	do
dump_socket_status	ed	else
enable_commands	environment	exec
explode	extract	file_name
file_size	filter_array	filter_mapping
find_call_out	find_living	find_object
find_player	first_inventory	float
for	foreach	function_exists
get_dir	glob	if
implode	inherit	inherit_list
input_to	insert_alist	int
interactive	intersect_alist	intp
living	localcmd	log_file
lower_case	m_delete	m_indices
m_sizeof	m_values	map_array
map_mapping	mapping	mappingp
member_array	mixed	mkdir

mkmapping	move_object	next_inventory
next_living	nomask	notify_fail
object	objectp	order_alist
parse_command	pointerp	present
previous_object	private	process_string
protected	public	query_host_name
query_idle	query_ip_number	query_ip_port
query_load_average	query_snoop	query_verb
random	read_bytes	read_file
regexp	remove_call_out	rename
replace	restore_object	return
rm	rmdir	save_object
say	set_bit	set_heart_beat
set_light	set_living_name	shadow
shared	shout	shutdown
sizeof	snoop	socket_accept
socket_acquire	socket_address	socket_bind
socket_close	socket_connect	socket_create
socket_error	socket_listen	socket_release
socket_write	sort_array	sprintf
sscanf	static	status
string	stringp	strlen
swap	switch	tail
tell_object	tell_room	test_bit
this_object	this_player	throw
time	trace	traceprefix
transfer	unique_array	upper_case
users	varargs	version
void	while	wizlist
write	write_bytes	write_file

B The Boot Sequence

When the MUD boots, several things happen in a certain order.

1. /obj/master.c is loaded. At this point, no include paths have been defined, so what include files master.c needs must be specified using absolute paths.

2. Include paths are now defined. /obj/simul_efun.c is loaded.

3. The objects listed in /room/init_file are loaded, in the order given in the list. This is typically some of the objects in the mudlib, like /secure/login.c, the daemons, the quest system etc. Usually, the various guilds are loaded. Then, the castles of the various wizards are loaded.

4. The game allows connections to be made and players can start to login.

C Tips, Traps and Trix

C.1 On the #pragmas strict_types and save_types

Using #pragma strict_types make life more interesting, which is part of an ancient Chinese curse. It is for those who code the objects of the mudlib, and those that like to type a lot of extra code, like to control their types, and those who think it looks nice (beware that beauty lies in the eye of the beholder). For most people, it isn't needed. The #pragma save_types are only of use in objects that are inherited a lot, like the lib objects.

C.2 Optimisation

As usual in optimisation, there are two things you must know before you start:

1. What to optimise, and
2. how to optimise that.

In general, you should use as good data structures as possible to save memory, and as good algorithms as possible to save CPU. Remember that there is an upper limit to how many evaluated nodes (evals) that can be used for a piece of code. Apart from that, it is really impossible to give other than general advice, since details on costs will change between driver versions. Here are some tips that will probably be valid even in the future, though:

- Avoid file accesses like the plague!
  Accessing the discs is magnitudes slower than accessing memory. Do not handle files unless you have to.
- Avoid nested loops.
  Using nested loops, it is very easy to hit the max evals.
- Avoid loops.
  In many cases, the loops can be deferred to the driver. The driver is *much* faster doing loops than LPC is. Typical examples are call_other() on arrays of objects, and sorting of arrays.
- Avoid expensive function calls in tight loops.
  Calls to functions has some overhead, especially if large data structures are sent to, and modified within, the functions.
- Optimise your loops.
  Be sure that the code you have inside a loop must be there. Move loop invariant code out of it.
- Mappings are sorted, 1.
  This means that looking up a key in a mapping is faster than searching for it in an array, even when using the efun member_array().

- Mappings are sorted, 2.
  For integers, the sort-order is recognisable to humans as ascending order. This can be used to sort an array by doing 'm_indices(mkmapping(arr,arr))'
- Mappings are sorted, 3.
  Using this fact with manufactured keys can be faster than using 'sort_array()'.
- Avoid subscripting, 1.
  Use 'foreach' whenever you can get away with it. It is the fastest loop construct available in LPC.
- Avoid subscripting, 2.
  Don't pick out single elements from large arrays or mappings. It takes forever.
- Avoid sprintf.
  Don't use 'sprintf()' when a a simple 'write()' will do,
- Use sprintf.
  Use 'sprintf()' instead of building presentation strings by concatenating things in loops.
- Use continue.
  Use 'continue' to short-circuit your loops as early as possible.
- Use 'break'.
  Use 'break' to exit loops etc. as early as possible.
- Use '||' operators to set defaults.
  The construction 'foo = foo || "default"' is faster than 'if (!foo) foo = "default", and just slightly less readable.
- Use 'switch - case' instead of 'if - elseif'.
  It is faster, and easier to read.
- Don't change size of your arrays.
  When you change the size of your arrays, new memory has to be allocated. If you do this more than once, you could consider allocating the array in advance using 'allocate()'.
- Use inheritance.
  Don't re-invent the wheel; if there is a lib object that you can use, do so. If you can use it with a few simple modifications, do that. If there isn't a lib object you can use, talk to an arch about it.

C.3 Three Keys to Successful Creating at NannyMUD

There are three things you need in order to have a successful career as a creating wizard at NannyMUD:

- Know thy LPC.
  Without the knowledge of how things happen in LPC, you will at best be able to fill in templates.

- Know Thy Lib.
  The lib contains a lot of standard objects. They offer a wide, and supported, selection of functionality and features. You need to get familiar with those in order to avoid re-inventing everything. If you do know the lib, you can make a passable area just by filling in templates.
- Know The Rules.
  Since NannyMUD has been around for a while, about every mistake has been done here. The rules collection reflects this by setting limits on things. Every rule is there because it is has been shown to be needed. You need to have some knowledge of them, or your creations will never make it into the game.

C.4 Some Common Mistakes

- Forgetting the inherited object.
  Often, people forget that they have inherited an object, and that the inherited object needs a call to reset(), too. Even worse is when you overload init(). Use the :: operator to call functions in the inherited object, or objects.
- Trusting results from efuns.
  Yes, at times efuns like environment(), this_player(), and even this_object() will return a zero, and rightly so. Beware, and be prepared to handle this.
- Doing 'call_other()' on a zero.
  Before using 'call_other', be sure that the object in which you call the function does exist. Also be prepared to have code to handle the case when it does not exist, in a meaningful way.
- Not initialising your arrays and mappings.
  Using the index operator, '[ ]', on a zero gives a run-time error. Remember that all variables are automatically initialised to zero, and that zero is not a valid array, nor mapping. Be sure that you do initialise them to ({ }) and ([ ]) respectively, at least.
- Providing the wrong number of arguments.
  When using the #pragma strict_types, you have to provide the right number of arguments when calling a function, unless it has been declared with the type modifier 'varargs'.
- Providing wrong types to operators.
  People will, sooner or later, call your functions with the wrong data type on the arguments. If your code relies on correct arguments being sent to it, rather than checking what actually were sent, it will break sooner or later.
- Exceeding index ranges.
  You should avoid writing code that will try to access arrays outside of it's index, *0..s-1*, where *s* is the number of elements in the array.
- Not exiting the function after destructing the object.
  When an object destructs itself, the statement to that effect should be followed by an immediate exit from the function.

- Trusting a player to be active.
  Beware that players can suddenly quit, go link-dead (LD), leave the location, die, etc. The LD case is especially obnoxious and hard to safe-guard against; all the others can be easily handled, and should.

- Using move_object() when you mean transfer().
  The efun transfer() should be used when you need checks for weights, i.e. when moving objects to and from players. To and from everything else, use move_object().

- Trusting player input.
  When the argument to your functions is based on player input, be sure to check that you get any, and check what you get. In particular, beware the player who on purpose gives negative arguments.

- Believing that the MUD is multi-threaded.
  The MUD is a single-threaded process: when your code is run, no other is.

- Using delay loops.
  Believe it or not, some people have actually created empty loops to create delays. Don't do that, use 'call_out' instead.

Index