DRAFT, August 2, 1999
Lingo in a Nutshell, published by O'Reilly & Associates

Please send feedback to lingonut@zeusprod.com

20

Lingo for C Programmers

This bonus chapter covers Lingo's internal operation, with an emphasis on comparing it to C/C++. The latest version of this document can be found at *http://www.zeusprod.com/nutshell/chapters/lingovsc.html* in Acrobat PDF format. This chapter is a superset of Chapter 4, *Lingo Internals*, in *Lingo in a Nutshell*. Even if you are not a C programmer, comparing Lingo to another language illuminates numerous issues. This chapter also offers additional details on Lingo and Director's development paradigm. You should be familiar with the first four chapters of *Lingo in a Nutshell*, which cover Lingo's foundation, before proceeding. You should also understand Director's Score metaphor as explained in Chapter 1, *How Director Works*, in *Director in a Nutshell*. Refer to Chapter 13, *Lingo Xtras and XObjects*, in *Lingo in a Nutshell*, and Chapter 10, *Using Xtras*, in *Director in a Nutshell* for details on extending Director (especially if you are a skilled C programmer).

Lingo Internals

There are four major issues for new Director users, especially those coming from other languages:

Performance
: Director is much slower than custom C applications. See the discussion below, and Chapter 9, *Memory and Performance*, in *Director in a Nutshell*.

The Cast, Score and Timeline Metaphor
: Director's Cast and Score affect your programming structure. Refer to Chapter 1, *How Director Works*, in *Director in a Nutshell* to understand the quirky interaction between Lingo and the Cast and Score, and how the timeline metaphor affects interactivity and events.

Media

Director's extensive use of media raises issues that are new to many programmers. Even perfect Lingo and Score usage can not overcome improperly prepared content. A Director programmer must know enough about content preparation to, at a minimum, instruct content providers in the proper preparation of video, sound and graphics. Refer to *Part III: Multimedia Elements* of *Director in a Nutshell*, especially Chapter 13, *Graphics, Color, and Palettes*, Chapter 15, *Sound and Cue Points*, and Chapter 16, *Digital Video*.

Interactivity

Director projects usually have lots of interactivity, and extensive GUI requirements. Refer to Chapter 2, *Events, Messages and Scripts*, Chapter 9, *Mouse Events*, and Chapter 10, *Key Events*, in *Lingo in a Nutshell*.

A client of mine, who is extremely proficient in C, and has a large library of multimedia routines written in C, complained that he had been corralled into using Director for a Macintosh kiosk project. Many of his objections were valid, especially given his C skills, existing C library, and lack of Director expertise. Lingo is comparable to English. If it is not your first language, English's spelling and pronunciation surely seem arbitrary. But if you are intimately familiar with English, it doesn't seem so bad. Once you learn Lingo's quirks, it is quite flexible and capable. For most projects, I've been much more productive with Director than I would have been with C, in part because my C skills are mediocre, but also because Director is geared towards multimedia development.

Director and Lingo offer the following advantages to most developers:

- Easy, fast, iterative prototyping while working with content providers or clients
- A graphical interface for easily constructing complicated sprite animations on Stage or in the Score
- Optimized blitting for fast compositing with ink effects
- Built-in transitions and palette controls
- A large number of third-party Xtras for specialized applications
- Excellent integration of media, such as graphics, sound, and video
- Re-usable cross-platform code across Macintosh and Windows
- Internet-based Shockwave delivery for major browsers, and internet-enhanced CDs
- Content is separate from Lingo code, allowing better collaboration
- An integrated environment with content development tools and extensive Lingo command set.

Most C programmers create one-frame Director movies, and do everything

manually via Lingo. They mistakenly avoid the Score even when creating animations, a task for which Director is ideal. Because they don't understand Director's paradigm, and haven't made allowances for the media requirements, they assume Director is hopeless. They tend to revert to C development, or complain vociferously about Director. The key is to distinguish between Director's deficiencies and one's own biases. This chapter, and the information in *Lingo in a Nutshell* and *Director in a Nutshell* should go a long way to helping you be more efficient, more quickly, in Director.

Most C developers equate Lingo with Director. Although Lingo has many deficiencies, the non-Lingo portions of Director, such as the Score, also have many strengths. C developers can create Xtras to extend Director's functionality to address its true deficiencies rather than its imagined ones.

I'll be the first to admit that Director is not ideal for the following tasks:

Fast-twitch video games (performance is too slow)

Real-time effects (overhead is often too great)

Low-end machine playback (RAM requirement is too great)

Computer Based Training (Macromedia Authorware is optimized for this)

That said, the above applications can be accomplished in Director by a skilled technician. There are additional applications, such as printing and device control, for which Director benefits from or requires third party Xtras. Refer to the resources mentioned in the *Preface* and throughout *Lingo in a Nutshell* and *Director in a Nutshell* for information on third-party Xtras.

The Paradigm and the Interface

Director's user interface is formidable, and a far cry from a simple text file in which a C programmer typically enters code. See the back cover of Director 6's *Using Director* manual for an overview of Director's windows (this is no longer included on the back cover of the D7 manuals). See *Part I: Director's Core Components* in *Director in a Nutshell* for details on all the Director windows and shortcuts. Refer to the *Glossary* in *Lingo in a Nutshell* for details, but here is a quick run-down of Director's terminology.

Score

Director's timeline (like a giant spreadsheet) for constructing cell animations. Refer to Macromedia's *Using Director* manual and to Chapter 1, *How Director Works*, and Chapter 3, *The Score and Animation*, in *Director in Nutshell*.

Cast

Director's database of all its assets. The Cast holds bitmaps, sounds, text, and even Lingo scripts. The term *Cast* (or castLib) has nothing to do with *casting* data between different data types, as is often done explicitly in C.

Movie
: A Director data file that contains the Score and an internal Cast. Movie files have nothing directly to do with QuickTime or other digital video formats.

Linking
: *Linking* means to attach an external asset (external sound or video, external castLib, or internet URL) to your Director movie. The term has nothing to do with *linking* in the C sense of creating an executable from object files.

Who's The Boss

C programmers may have a hard time letting Director handle so much of the action. If C is like a Ferrari with a manual transmission, Director is more like a couch on wheels with an automatic transmission, cruise control, and a chauffeur. James Terry of *Kandu* (*http://www.kandu.com*) writes:

> I'm not sure my programming style changed [when I moved from C to Director] (maybe it should have), so much as my mental picture of the "program." In C/C++ everything is there in the code and I feel in complete control. When using Director it's more like working with a partner who is in charge of some parts. I don't have that same feeling of control. Of course it's nice that the partner can do some work for you. The whole point of working with Director is that you're just part of the team; knowing how Director is working is so crucial. The biggest adjustment I had to make was just plain finding my source code! As a C/C++ programmer I was used to having all my source code in easily viewable text files.

If you sympathize with James you should read *Where the Hell Are My Scripts?* in Chapter 2, *Events, Messages and Scripts*, in *Lingo in a Nutshell*. In Director, your scripts are stored in script cast members that reside in the Cast window like any other asset. Prior to D7, each script was limited to 32 KB of text, but you can have a virtually unlimited number of scripts.

Memory, Pointers and Memory Access

Director's memory usage is dominated by multimedia content, primarily graphics, rich text and non-streaming audio. In Lingo, you don't explicitly allocate RAM, as with *malloc* or *alloc* in C. There is no low-level access to RAM, or to Director's off-screen buffer, except via Xtras. Nor are there *address* or *indirection* operators, as in C, that would give access to a variable's memory space. In Lingo, items are disposed of by setting them equal to zero, except for XObject instances which are freed with *mDispose* (don't use *mDispose* with Xtras). When an item in memory is no longer referenced by any variables, Director will deallocate and recapture the memory (i.e. *reference count-based garbage collection*), but on an ambiguous schedule. Refer to Chapter 9, *Memory and Performance*, in *Director in a Nutshell* for details on allocating and freeing memory, loading and unloading assets, and Director's memory requirements. Lingo does not support a *sizeof* function, because the size of various data types

is not implementation-dependent, as in C.

Compilation, Interpretation and Performance

Lingo is not *compiled*, but rather *tokenized* into *byte-code*. The byte-code is the same on both the Macintosh and Windows platforms, and is translated at runtime by the Projector (or the Shockwave plug-in) into machine-specific instructions. This is convenient for cross-platform development, but does not offer the speed of assembler. Because Lingo is not compiled, there is no separate compiler, or compiler directives.

Director 7 supports a new "dot syntax" similar to C or JavaScript, as well as the old Director 6 Lingo syntax. There is no performance difference between the two. They both "compile" to the same byte-code.

Director 6's `Modify Movie Properties` *Allow Outdated Lingo* option controls how Director interprets the Lingo of movies created in previous versions of Director. The checkbox is selectable only if a movie has been upgraded from a prior version of Director (this option does not appear in Director 7). If you are updating movies from older versions of Director (especially from Director 3.1.3 to 4.0, and Director 4.0 to 5.0) refer to Tables 4-1 and 4-3 in Chapter 4, *CastLibs, Cast Members, and Sprites*, in *Director in a Nutshell*. Your main concern when upgrading versions is not necessarily the new Lingo features, but rather the changes in the behavior of existing functions that might break legacy code. Such a change from version to version would be exceedingly rare in a language such as C, but is common in Director. In Director 6, the major alteration from prior versions involves sprite message passing. Refer to Chapter 2, *Events, Messages and Scripts*, and Table 17-2 covering Lingo changes from Director 5 to Director 6, in *Lingo in a Nutshell.* The most notable change from Director 6 to Director 7 is that using parentheses around a *member* expression is no longer supported.

Don't use this:

```
member (x) of castLib y
```

Use either of these instead:

```
member (x, y)
member x of castLib y
```

A second notable change is that a period (.) can no longer be used in a symbol or variable name, because it confuses the parser, and is instead used to specify a property of an object, member or sprite, such as:

```
put sprite(5).locH
```

A third notable change is that D7 no longer allows undeclared local variables as was possible in D6, as described under "Special Treatment of the First Argument Passed" on page 61 of *Lingo in a Nutshell*.

For more details on Lingo changes from Director 6 to Director 7, see the Director 7 ReadMe file. See also *http://www.zeusprod.com/nutshell/d7diffs.html* and *http://www.zeusprod.com/nutshell/dotsyntax.html* (both under construction).

Lingo executes egregiously slowly compared to C. You must be much more conscious of both media requirements and program execution times, even for simple things like loops. There is no compiler optimization. You must manually optimize your code, such as by moving assignment statements outside of *repeat* loops.

The performance of your Lingo code may be dwarfed by the time required to load media assets or download Shockwave data, but when performing an operation thousands of times, you cannot ignore Lingo's execution speed. Table 20-1 shows the comparable speed of 10,000 iterations of a various Lingo commands performed in Director 6 using a repeat loop of the form:

```
repeat with x = 1 to 10000
  statement(s)
end repeat
```

These tests were run on a 33 MHz Macintosh 68040 processor, which is very slow by today's standards. But similar tests would run at least an order of magnitude faster on the same CPU if written in C. Lower numbers are faster, but the numbers should be used for comparison purposes only. Remember that the absolute speed difference among Lingo commands is rarely noticeable for a single execution, and that Table 20-1's differences are amplified by performing an operation thousands of times.

Note especially how slow the *put* statement is when used to output text in the Message window. Note that even the *nothing* statement takes time to execute, and that the *case* statement can be slower than the corresponding *if* construct. Note that floating-point math is slower than integer math and that text handling tends to be very slow.

The speed of an operation may vary across different versions of Director and will vary across platforms and in Shockwave. For example, checking the value of *the ticks* is noticeably slower in D7 than in D6. The speed of an operation may be affected by the speed with which cast member references within an expression are resolved, or the speed of cast member loading See *Access Speed and Name Caching* in Chapter 4, and see "Gauging Performance" in Chapter 9 in *Director in a Nutshell*.

Table 20-1: Comparative Lingo Execution Speed

Operation	**Lingo Code within repeat loop**	**Comparative time**
Repeat Loop	`--`	0.32

	No code, only a comment inside the loop	
No operation	`nothing` (Lingo's no-op statement)	0.47
Assignment	`set y = 5`	0.37
Command execution	`testCommand`	0.72
Function call returning value	`testFunction()`	0.92
Print in Message window	`put x`	5400.0
Text Parsing	`set y = word 3 of "this is text"`	2.18
Type Conversion	`set y = integer (5.5)`	2.05
Integer Math	`set y = 5 * 12`	0.4167
Floating Point Math	`set y = 5.0 * 12.4`	1.3667

If statement	`if x > 5000 then`
	`  set y = 5`
	`else`
	`  set y = 4`
	`end if`
		0.50

Case statement	`case (TRUE) of`
	`  (x > 5000):`
	`  set y = 5`
	`  otherwise`
	`  set y = 4`
	`end case`
		.617

Logical *AND*	`if (x > 5000) AND (x < 8000) then`
	`  set y = 5`
	`end if`
		.567

Instantiate script by number	`set y = new (script 233)`	3.6
Instantiate script by name	`set y = new (script "Birth test")`	7.92[1]

1, This test was performed with the parent script in castmember slot 233, If the parent script was in the first castmember slot, the comparative time was only 6.35.

Dynamic Compilation

Throughout this book, I use the term *compile*, as it's used by Macromedia, to indicate the tokenizing of a script. When you recompile a script, Director only checks the syntax, and for uninitialized local variables. It does not perform any type checking, nor validate the number of parameters to a function call. There is

no Lingo equivalent to C's *lint* utility, which warns about global variables that are also declared as locals, unused local variables, and similar potential errors.

Director automatically compiles any uncompiled scripts when you play your movie. There is no lengthy recompilation of scripts that haven't changed.

There is no linking of object modules and libraries into an executable, so Director will not warn you if you refer to a non-existent function. An error will simply occur at run-time if a function cannot be found.

In Director, the term *linked* refers to an external cast library or an external asset (such as a bitmap, sound, or video). Xtras are in some ways analogous to DLLs or shared libraries that exist in C programs.

Lingo gives you the convenience of, but is somewhat faster than, an interpreted language such as BASIC. The same Lingo can be run on multiple platforms, because it is not stored in a machine-specific format (i.e. assembler). Unfortunately, you pay for the development benefits with slow runtime performance.

It is not unusual for Director to lose track of the compiled version of a script, and you should recompile using `Control R` or `Recompile All Scripts` to make sure your Lingo is properly compiled. See "Compiling Scripts" in Chapter 2 of *Lingo in a Nutshell* for details on compiling scripts and on addressing problems with corrupted scripts.

Lingo's dynamic compilation allows you to rapidly change and test your Lingo. Lingo can even be modified or created at runtime by setting *the scriptText of member* property or using the *do* or *value* command. Even the base class of an object can be changed dynamically by setting the *ancestor* property of a Behavior or Parent script. See Chapter 12, *Behaviors and Parent Scripts*, in *Lingo in a Nutshell*.

Pre-Processor Directives

There is no pre-processor phase to Lingo compilation; therefore, Lingo does not support any of C's pre-processing directives such as `#define`, `#ifdef`, and `#include`. You can simulate pre-processor directives using a constant, global variable, or system property as shown in Example 20-1.

Example 20-1: Simulating Pre-Processor Directives

```
on startMovie
  global debug
  if the runMode = "Author" then
    set debug = TRUE
  else
    set debug = FALSE
  end if
```

```
end startMovie

on someHandler
  global debug
  if debug then
    put "Debugging statement for author-time only"
  end if
end someHandler

on anotherHandler
  if FALSE then
    -- This block of code will never be executed
    put "Block of code commented out"
  end if
end anotherHandler
```

Use *the platform* property to determine whether you are running under Macintosh or Windows as shown in Example 8-1 in Chapter 8, *Projectors and the Runtime Environment*, of *Director in a Nutshell*.

Language Definition and Programming Structure

C's command set is exceedingly simple, and its syntax and structure are uniform, efficient, and elegant. By comparison, Lingo is lumbering and wildly inconsistent. Director 7 introduces the dot syntax and bracket syntax for properties and lists, so at least Lingo is not as verbose as in prior versions.

For example, in Director 6 you'd use:

```
put the width of member 5 of castLib 1
put getAt (the actorList, 1)
```

In Director 7, you can instead use:

```
put member(5,1).width
put actorList[1]
```

Whereas C provides very low-level access, Lingo provides high-level commands to play sounds or download bitmaps. Whereas C's command set is limited to a few dozen keywords, Lingo has over 1000 commands and properties specific to media, the GUI, and event processing. Lingo incorporates many of the support functions ordinarily found in C libraries, such as *stdio*.

A C program is roughly equivalent to a Director "project," made up of multiple movies with multiple scripts, handlers, or functions. Table 20-2 shows some of the syntactical differences between Lingo and C.

Lingo handlers are declared using the *on* keyword, but there is no parameter type-checking or return value type-checking associated with a function declaration or separate function prototype. Lingo uses the keyword *the* to identify properties, such as *the clickOn*, and *the mouseH*.

Table 20-2: Syntactical Comparison Between Lingo and C.

Operation	**C**	**Lingo**
Handler declaration	*handlerName* (*a*, *b*, *c*) {*statements;* }	on *handlerName a*, *b*, *c* *statements* end
Event Loop or Messaging Loop	Implemented by programmer	Implemented by Director which automatically dispatches events.
Entry point	main()	N/A. Events are dispatched by Director to various scripts.[1]
Comment characters	`/*` and `*/`, or `//`	`--` (two hyphens)
Continuation character	None. Semi-colon (;) terminates a statement	Carriage return terminates a statement unless line ends with the Lingo continuation character (¬) created with Option-L or Option-Return (Mac) or Alt-Enter (Win)
Index into an array (referred to as a "list" in Director)	array[*index*] or use pointer math. *Index* is zero-relative.	Director 6: getAt(*list*, *index*) Director 7: list[*index*] No pointer math. *Index* is one-relative.
Element in a structure or list	struct.element *struct->element	Director 6: the *property* of *object* Director 7: object.property
Variable-length argument lists	*varargs* macro	*the paramCount* and *param(n)*. See Example 1-38 in *Lingo in a Nutshell*.
Size of data types	*sizeof()* function	Data type sizes are fixed. See *length()* function for strings and *count()* for lists.
Keywords	Cannot be used as variable or handler names.	Most properties include the keyword *the*. Variables and handlers can use the same name as Lingo keywords, but this should be avoided for clarity.
Case-sensitivity	Strictly case-sensitive.	Generally case-insensitive. See *Appendix C* in *Lingo in a Nutshell* for exceptions.

1. Initialization code can be placed in your *prepareMovie* or *startUp* handlers, which are

generally called when a movie or Projector starts. See Chapter 2 of *Lingo in a Nutshell*.

Table 20-3 compares more of C's and Lingo's features.

Table 20-3: Operational Comparison Between Lingo and C.

Item	**C**	**Lingo**
Compilation	Compiled into native assembler	Tokenized into platform-independent byte-code
Object modules and libraries for linking	Supports *.o* and *.lib* files	Uses external cast libraries to hold shared content or Xtras to extend functionality.
Preprocessor directives	Supports #define, #ifdef, #include, etc.	Not supported. Simulate as shown in Example 20-1.
Code speed and memory usage	Extremely small and fast	Very slow and fairly large, although dwarfed by media.
Multimedia support	None, but usually provided via libraries	Extensive cross-platform support, optimized blitting.
Command set	Limited, low-level	Extensive, high-level
Data Types	Extensive built-in and user-definable data types. Strongly typed.	Limited data types, complex objects, and obscenely loose data typing and conversion.
Parameter type-checking	Automatic via function prototyping	Must be done manually using *the paramCount*, *integerP*, *floatP*, etc. See Example 1-39 and Chapter 5, *Data Types and Expressions* in *Lingo in a Nutshell*.
Parameter passing	Arrays (including strings) are always passed by reference. Can force pass by reference for other data types.	Lists and objects[1] are passed by reference. Other types, *including strings*, are passed by value. Can't force pass by reference.
Compound expression evaluation	Evaluates only clauses as necessary	Evaluates all clauses of expressions using *and* and *or* even when not necessary.
Math operations	Extensive, including bitwise operations and complex (imaginary) numbers.	Limited. No support for complex (imaginary) numbers. D6 and prior require Xtra for bitwise operators. D7 has undocumented *bitAnd*, *bitNot*, *bitXor*, and *bitOr* operators.
Math Precision (see *Math Precision* below)	Double-precision floats and long integers	See *the maxInteger*. Float precision to 15 decimal places. Display precision indicated *the floatPrecision*.

Hexadecimal and Octal number support	Yes	No. although Hex RGB colors can be specified for D7 color objects. See *http://www.zeusprod.com/nutshell/download.html* for a utility to convert between different number bases (under construction).

1. See "Special Treatment of the First Argument Passed" in Chapter 1 of *Lingo in a Nutshel*l for important details. See also Chapter 6, *Lists*, in *Lingo in a Nutshell*.

Macros and In-Line Code

Director does not support macro definitions, or in-line macros, as with C. Subroutine calls takes several orders of magnitude longer in Lingo than in C, so you should code in-line when performance is crucial. The `File Save as Java` option supported in D6.5 and D7 does support in-line Java calls within your Lingo code.

Header files

Lingo does not support header files (i.e. include files typically ending in .h), as does C. This is not necessarily an issue, as Lingo does not require lengthy declarations, except perhaps for global variables, which must be re-declared manually in all scripts that use them. Put any common scripts in an external cast library that can be attached to any Director movie.

Objects and OOP

In Director, an *object* is an instance of a Parent script or Xtra, or other complex data structure, including lists. It is not an "object module" resulting from compilation, as in C. Director has had decent OOP support since version 4, but it differs from OOP in other languages. Most notably, child objects name their ancestors, rather than vice-versa, and the base class can be changed dynamically. Refer to Table 20-12, and to Chapter 12, *Behaviors and Parent Scripts*, in *Lingo in a Nutshell*.

Director 7 introduces two new types of objects: date objects and color objects as described in the D7 documentation. Refer also to the *ilk()* function discussed in Chapter 5 of *Lingo in a Nutshell*.

Libraries and DLLs

Director requires external procedures to be packaged within Xtras (which are accessed dynamically at run time) and does not make use of C-style library (*.lib*) files or object (*.o*) files. (Remember, in Director, *linking* means to attach an external cast library to a Director movie, not to create an executable from library and object modules.) Put any common scripts or media in an external cast library that can be attached to any Director movie.

Unlike C libraries in which only the required functions are included in the executable, all Lingo scripts in the current movie and any linked casts are *always* loaded into RAM, whether they are needed or not. They are only unloaded until a movie ends. Avoid excessive numbers of scripts, or modularize your movies and casts libraries containing scripts to minimize RAM usage.

Custom extensions to Director are implemented (in C) as Xtras. Refer to Chapter 10, *Lingo Xtras and XObjects,* in *Lingo in a Nutshell*.

Executables

Director movies are not necessarily compiled into an executable. They can be incorporated into a *Projector* (a platform-specific runtime engine) but can also be left external for easy updating. Use `File ➪ Create Projector` to create your Projector(s). You must purchase the Director authoring tool for each platform for which you intend to release a Projector. See Chapters 4 and 8 in *Director in a Nutshell* for many more details.

Assignment, Math and Logical Operators

Lingo supports or can simulate most of the assignment, math, and logical operators supported in C. The syntax of the two languages is identical for those operations not shown in Table 20-4, such as multiplication, addition, subtraction and division.

Lingo does not support complex numbers. Prior to Director 7, it does not support bitwise operators without an Xtra or other custom code, and even in D7 the bitwise operators are undocumented and unsupported.

The *log()*, *exp()*, and *sqrt()* functions are the same in Lingo and C, but *power()* is Lingo's equivalent to C's *pow()* function. The *sin()*, *cos()*, *tan()* and *atan()* functions are the same in Lingo and C, but *asin()*, *acos()*, *sinh()*, *cosh()*, *tanh()*, *asinh()*, *acosh()*, and *atanh()* are not supported by Lingo. (See *http://www.zeusprod.com/nutshell/examples.html* for derivations of the arcsine, arccosine, and the hyperbolic trig functions using the built-in Lingo trig functions.) Imaginary numbers are not supported directly in Lingo.

Table 20-4: Math and Logical Operators in C and Lingo.

Operation	**C**	**Lingo**
Assignment[1]	*x* = 5;	Director 6: set *x* = 5 set x to 5

		put 5 into 5 Director 7: *x* = 5
Comparison	if *x* == 5 then...	if *x* = 5 then...
Modulo division	*x* % *y*;	*x* mod *y*
Modulus and assignment[2]	*x* %= *n*;	set *x* = x mod *n*
Increment[3]	*x*++; or ++*x;*	set *x* = *x* + 1
Decrement[3]	*x*--; or --*x;*	set *x* = *x* - 1
Raise to an exponent	pow(*x*, *y*)	power (*x*, *y*)
Square of a number	sqr(*x*)	power (*x*, 2)
Logical *and*	&&	and
Logical *or*	||	or
Logical *exclusive-or*	no built-in operator	See Example 5-4 in *Lingo in a Nutshell*.
Logical *not*	!	not
Not equal to	!=	<>
Bitwise *and*	&	bitAnd[4]
Bitwise *or*	|	bitOr[4]
Bitwise *exclusive-or*	^	bitXor[4]
Bitwise *not*	~	bitNot[4]
Bitwise *shift-right* and *shift-left*	>> <<	no Lingo equivalent

Conditional assignment	*var* = (*expr*) ? *val1* : *val2*;	`if (expr) then`
		`  set var = val1`
		`else`
		`  set var = val2`
		`end if`

1. C programmers will often forget to use the *set* keyword when assigning variables. Prior to D7, a statement such as *x = 5* will cause a, *"Misplaced Operator,"* error.

2. C's `*=`, `/=`, `+=`, and `-=` operators are not supported in Lingo, but can be simulated in an analogous manner.

3. C will increment an array pointer by the size of the data type. Lingo does not support pointer arithmetic.

4. Director 7 only (unsupported and undocumented). In D6, use Xtra or custom Lingo.

Math Precision

C supports signed and unsigned short, standard, and long integers and both single- and double-precision floats. C's native float data type is double-precision, which use 8-bytes under Windows and 10-bytes on the Macintosh. Lingo uses 4-byte integers (whose range, as indicated by *the maxInteger*, is 2^{31}-1). *The floatPrecision* property controls the *display* of floating-point numbers, whose accuracy is approximately 15 decimal places. Refer to Chapter 8, *Math and Gambling*, in *Lingo in a Nutshell* for details on math precision and overflow conditions.

Loops, Flow Control and Multi-line Statements

Lingo and C provide similar constructs for looping and flow control, but the keywords and syntax are somewhat different. Whereas C generally uses curly braces ({}), Lingo uses separate keywords to indicate the beginning and end of multi-line structures, as shown in Table 20-5. In C, you can declare a local variable within a code segment offset by curly braces, and it will have scope only within the braces. Lingo supports scoping within a handler, but not subscoping within a multi-line structure. You should declare your variables as described in Chapters 1, 3, and 12 of *Lingo in a Nutshell*.

As does C, Lingo re-evaluates the conditional expression each time through the loop. Therefore, any changes made to the index variable in a loop affect the number of loop iterations.

Lingo will evaluate all clauses within a logical expression whether it needs to or not. This is inefficient, and differs from C, which only evaluates the minimum clauses required to evaluate the overall logical expression. Lingo's insistence on evaluating all clauses may cause errors in code that relies on C's style of expression evaluation. Refer to *Boolean Expressions and Logical Operators* and especially Example 5-3 in *Lingo in a Nutshell*.

Table 20-5: Flow Control Structures Lingo vs. C

Operation	**C**	**Lingo**
multiple-statement structures	{}	on...end, if...end if, case...end case, repeat...end repeat, tell...end tell, beginRecording...endRecording
Loop while an	`while (`*`expression`*`)`	`repeat while` *`expression`*

expression is TRUE	`{statement;}`	`statement` `end repeat`
Loop, checking condition at end of loop	`do {statement;}` `while (expression);`	No exact equivalent. Simulate with standard *repeat...while* loop.

Loop for a specific number of iterations	`for (x=1; x++; x< n)`
	`{statement;}`
		`repeat with x = 1 to n`
		`statement`
		`end repeat`

Loop through items in a list	`for (x=0; x++; x< n)`
	`{y = array[x];}`
		`repeat with y in list`
		`statement`
		`end repeat`

Next loop iteration	continue	next repeat
Abort loop	break	exit repeat

Switch or case statement [1]	`switch (expression) {`
	`case value: statement;`
	`break;`
	`default: statement;}`
		`case (expression) of`
		`value: statement`
		`otherwise: statement`
		`end case`

Quit the program	exit	quit or halt
Go to code label	goto	No equivalent. See Table 20-11.
Abort the call stack	No equivalent	abort

1. Lingo execution does not “fall through” from one clause of the case statement to the next, despite lack of a C-like *break* statement.

Parameter and Variables

Lingo handles parameter passing and variables very different than in C and most other languages.

Passing Parameters

In Lingo, most data types, including strings, are passed by value, except lists (and other objects) which are passed by reference. Therefore, changing the contents of a list within a handler will change the list in the calling routine. See Chapter 6, *Lists*, in *Lingo in a Nutshell* for more details. Similarly, you cannot change individual elements of *the vertexList* property of vector shape members in D7, because Director operates on a copy of *the vertexList*. You must reassign the entire *vertexList* at once, or use the vector shape functions *addVertex()*, *moveVertex()*, etc.

In C, arrays (including strings, which are arrays of chars) are passed by reference by default. You can pass the address of (i.e. a pointer to) a variable, so that a handler can modify multiple variables. Lingo doesn’t support indirection,

so non-list data types can only be passed back to the calling routine using a *return* statement (or accessed indirectly via a *global* declaration). In Lingo, you can instead pass multiple parameters in a list, which can then be modified individually and passed back via the list structure.

In Lingo, the parentheses surrounding the arguments to a function are mandatory only when that function returns a value. Unlike C, the parentheses are optional when the calling routine is not receiving a return value.

In Lingo, if the calling routine omits an argument to a function, the corresponding parameter inside the function is set to `VOID.` In C, an error would be generated by the pre-compiler, or the omitted argument would result in the corresponding parameter holding a meaningless value.

Lingo allows nested function calls, as does C, such as:

```
set x = random (max(y, z))
```

Unlike C, Lingo can *not* perform a variable assignment within a compound expression (recall that prior to D7, Lingo assignment required the keyword *set*). Although the = operator is used for both assignment and comparison in Lingo, within a compound expression it is always treated as a comparison operator:

The following would be evaluated differently in Lingo than in C:

```
if x = 1 > 0 then...
```

Assuming it was converted to C syntax, C's precedence would cause it to be interpreted as follows:

```
if x = (1 > 0) then...
```

The variable *x* would be set equal to the result of the expression (1>0).

In Lingo it would be evaluated differently. First of all, it would cause a syntax error unless *x* had previously been declared as a global or property variable, or used as a local variable. If *x* was already declared, the expression would be evaluated as:

```
if (x = 1) > 0 then...
```

In Lingo, *x = 1* would be evaluated first and the logical result would be compared to 0. Note that in C, the >, <, >=, and <= operators have precedence over = and ==. In Lingo, the = operator has the same precedence as <, >, <>, >=, and <=, and these operators are evaluated left to right in the order in which they are encountered.

Refer also to "Parameters and Arguments" starting on page 55 in Chapter 1 of *Lingo in a Nutshell*.

Lingo variable and function names are case-insensitive, and can be exceedingly long (about 250 characters). In C, variable and function names are strictly case-sensitive. See Appendix C in Lingo in a Nutshell for situations in which Lingo is case-sensitive.

Declarations, Type Checking and Conversion

In comparison to C, Lingo's data typing is loose to the point of being obscene. Lingo never explicitly declares a variable's data type, nor a function's return type, as opposed to C which is strongly typed. A Lingo variable's data type is implicitly defined by the type of data assigned to it, and can be changed by simply assigning a new datum to it.

Data Types

Except for lists (and objects), all Lingo data types, including strings, are passed by value not by reference. There is no exact Lingo equivalent for many C data types, but Table 20-6 shows the approximate Lingo and C data type equivalencies. In C, the storage requirements for each data type are machine-dependent—the *sizeof()* function indicates the number of bytes required for the item. In Director, the size of each data type is fixed, not machine-dependent. Note that Lingo does not support *typedef* for creating your own data types.

While a variable exists, even if it is set to `VOID`, it always requires at least 8 bytes:

Local variables (those used within a single handler) are allocated when the handler is called and freed when the handler terminates.

Property variables are allocated when the object, such as a Behavior script, Parent script or Xtra instance, is instantiated. They are freed when the object is disposed of (see below).

Global variables persist for the life of Director or the Projector. Using *clearGlobals* will free the storage used for complex data types, but the global itself will still occupy 8 bytes.

Symbols persist for the life of Director or the Projector.

String, integer, and floating-point constants are freed after the handler in which they are used terminates.

Child Objects and Xtra Instances

Every instance of a Parent script, Behavior, or Xtra occupies memory until it is no longer needed. Avoid instantiating thousands of objects, and clear any objects once you are done using them so that Director can reclaim their memory. Avoid two objects pointing to each other or objects that have a reference to themselves. When you delete an object, make sure that any properties that it has that are references to other objects, are also set to VOID or zero.

Table 20-6: Lingo and C Data Type Comparison

Data Type	C	Size in C	Lingo	Size in Lingo
array of elements	array	Dimension * bytes per element, or *malloc*'ed RAM	list	sum of sizes of all elements[1]

single character	char	1 byte	string	8 bytes + 1 byte per character
character string	char *	array of 1-byte elements	string	8 bytes + 1 byte per character[1]
enumerated data type	enum	*n* * bytes per element	property list	sum of elements[1]
floating-point number[2]	float, double, long double	4, 8, or 10 bytes on PC, 8 or 10 bytes on Mac	float	16 bytes
integer[3]	short, int, or long	1 byte, 2 or 4 bytes, and 4 bytes	integer	4 bytes
structure	struct	sum of elements	property list, or properties of an object	sum of elements[1]
Complex objects, such as child objects	struct	sum of elements	object	sum of size of properties, plus headers (varies for each type)
pointer	*	4 bytes	object	8 bytes
void data type	void	4 bytes	See VOID constant	4 bytes
media	array	dimensioned by *malloc*	picture, media, script, sound	Depends on media type, dimensions, sampling rate, etc.
symbol	N/A	N/A	symbol	8 bytes

1. Lingo lists and strings are allocated space dynamically as needed.

2. Lingo floating-point precision is 15 decimal places. *The floatPrecision* property affects only the display precision, not the calculation precision. See Chapter 8 in *Lingo in a Nutshell*.

3. Lingo does not support separate types for *long* (long integer), *short* (short integer), *unsigned int*, *unsigned long*, and *unsigned short*. A Lingo *integer* is similar to a C *long*. *The maxInteger* property returns an integer's maximum range (+/- 2^{31}-1)

Data Type Checking

Lingo does not verify the number or type of arguments to a function call, nor does it support function prototyping, as in C. In C, because every variable's type is fixed, it is assumed that you know a datum's type. Because Lingo is so loosely typed, it provides functions—most notably *ilk()*, *integerP()*, *floatP()*, *stringP()*, *objectP()*, and *voidP()*—to determine a datum's type. Refer to Tables

5-3 and 5-4 in Lingo in a Nutshell. Refer to the Example 1-39 in *Lingo in a Nutshell* for techniques for verifying parameters to a function manually.

Data Type Coercion and Conversion

Lingo casually converts between various data types with sometimes puzzling results, as opposed to C-style coercion and conversion which is usually explicit, although C performs automatic conversion when passing certain data types to functions (see Table B-3 in Appendix B of *Practical C Programming* by Steve Oualline, published by O'Reilly and Associates).

Refer to "Type Assignment and Conversion" in Chapter 5 of Lingo in a Nutshell. Lingo and C use different syntaxes for converting and coercing different data types, as shown in Table 20-7. Note that in C, one needs to know the original type of the datum that is being converted in order to select the proper function. C also supports type casting, which has no direct analog in Lingo besides actual type conversion. In C, a single memory address can be accessed with pointers of differing data types, which would allow it to be accessed, say, in either 1-byte or 4-byte chunks. There is no need to explicitly convert a single ASCII digit to an ASCII char in C—the datum can simply be coerced to a `(char)`.

Table 20-7. Data Type Coercion and Conversion in Lingo and C

Coerce to	**C**	**Lingo**
integer	(int), *ftoi()*, *ftol()*, *strtoi()*, *strtol()*, *strtoul()*, *atoi()*, *atol()*	*integer()*
float	(float), (double), *atof()*, *itof()*, *strtod()*, *strtof()*	*float()*
character or string	(char), (char *), *itoa()*	*string()*
ASCII character	(char)	*numToChar()*
ASCII code	(short)	*charToNum()*

Lingo's *value()* function does not technically perform type conversion. It actually invokes the Lingo interpreter to evaluate a string. It usually evaluates strings as if they are variable names and evaluates numeric characters as either integers or floats.

Data Storage Classes and Scope

Lingo property variables and global variables are persistent, in that they maintain their values even when the current handler is exited. Lingo does not support static function declarations. All functions within a Score script or Parent script are implicitly static (hidden from other scripts). Handlers in Movie scripts

are never static.

Table 20-8 compares Lingo and C data storage classes. There is no Lingo equivalent for the *register* (register variable) C data storage class.

Table 20-8: Lingo and C Data Storage Classes.

Operation	**C**	**Lingo**
global variable	extern or static	global
local variable	auto	no declaration needed
property variable	static	property
Declare a constant	#define *constant*	No true equivalent.[1]

1. Director pre-defines the constants `BACKSPACE`, `ENTER`, `EMPTY`, `FALSE`, `QUOTE`, `RETURN`, `TAB`, and `TRUE`, plus `PI`, `SPACE`, and `VOID`, which were added in Director 6. Define string constants using *#symbolName*. See Symbols" in Chapter 5 and also Chapter 19, *The Lingo Symbol Table* in *Lingo in a Nutshell*.

Lists and Arrays

Lingo "lists" are roughly equivalent to arrays in C and Table 20-9 compares the two. Unlike C arrays, Lingo lists are allocated automatically, and can change size arbitrarily. There is no dimension or *malloc* statement in Lingo, and a single list can contain data of various types *simultaneously*. A two dimensional array is implemented as a list of lists in Lingo, but it cannot be referenced as a one-dimensional array, as is often done in C. Using an index which exceeds the length of the array can trample memory is C, or return a meaningless value, without warning. In Lingo, it would either generate an error, or cause the list to be lengthened. Unlike most Lingo data types, lists are passed by reference, not value. See Chapter 6, *Lists*, in *Lingo in a Nutshell* for more details. Note that strings (discussed below) are a separate data type in Lingo.

Table 20-9: Arrays in C versus Lists in Lingo.

Array Operation	**C**	**Lingo**
Allocation	Dimensioned when declared, or allocated with *malloc* or *alloc*	Allocated automatically as elements are added or deleted.
Array indices	Zero-relative	One-relative
List elements	All elements must be of the same type	A single list can hold elements of different data types.[1]
Using index beyond last element's index	Reads or writes past end of allocated array without warning	Error is generated when reading past last element. List is lengthened if adding

	(may be a compiler option).	elements or characters past end.
Declaring a two dimensional array	Specify list of pointers, and dimension at least one index	Specify a list of lists, or use a property list (2x1 array). No need to predefine dimensions.
Accessing two dimensional array	Can be accessed with a single index	Must be accessed as list of lists.
Passing as a parameter	Passed by reference	Passed by reference

1. All elements in a linear list must be simple elements, whereas all elements in a property list must be property:value pairs.

In Director 7, lists can be accessed with C-like syntax. For example, the first element of a list named *numbers* could be accessed as:

```
numbers[1]
```

Likewise, a list of lists in D7 can be accessed as would a two-dimensional list in C. The third element of the fifth sublist would be accessed as:

```
myList[5][3]
```

Strings

In C, a string is not a built-in data type; it is implemented as any array of type `char`, with a null character (ASCII 0) as a terminator. C strings can be manipulated as one would manipulate any C array. In Lingo, a string is a unique data type, and there is no single-byte data type, such as C's `char` type. When working with Lingo strings, you need not allocate a specific number of bytes, nor worry about null-termination. A Lingo string with no characters has zero-length, as reported by the *length()* function. In C, a string with no characters, also has zero length, as reported by *strlen()*, which doesn't count the null-terminator. Lingo strings are allocated automatically and dynamically. Whereas a Lingo field is limited to 32000 characters (31.25 KB) prior to D7, a string variable can be much larger. I created a string with a length of almost 2 MB. Unfortunately, because Lingo strings are passed on the stack by value (which is very inefficient), attempting to call the *length()* function with such a large string can overflow Director's stack. The length of a C string (array of `char`) is limited only by available RAM. Because C strings are passed by reference like any other array, even exceedingly large C strings will not overflow the stack.

> Passing strings on the stack by value is one reason Lingo's text processing is egregiously slow.

You cannot directly modify a string argument to a Lingo function. Instead, you must modify a copy of the string, and pass it back as a return value. Table 20-10 compares strings in C and Lingo. Refer to Chapter 7, *Strings*, in *Lingo in a Nutshell* for many more details on string manipulation in Lingo, especially concatenation. See Chapter 10, *Keyboard Events*, in *Lingo in a Nutshell* for

details on user input of strings. See Chapter 14, *External Files*, in *Lingo in a Nutshell* for details on reading and writing strings using external files. See Chapter 12, *Text and Fields*, in *Director in a Nutshell* for details on displaying strings on Stage in #text, #richText, or #field cast members.

Table 20-10: Character Arrays in C versus Strings in Lingo.

Array Operation	**C**	**Lingo**
Allocation	Allocated as an array of `chars`	Allocated dynamically as data is added to string.
String manipulation	Accessed as an array of `chars`, or with common library functions	Accessed with numerous Lingo commands.
Null String	Single terminating character (ASCII 0)	EMPTY
Access single character	Access as with any array, by index	char x of string chars x to y of string
Accessing character beyond string length	Reads or writes past end of allocated array without warning.	EMPTY is returned when reading past last character. List is lengthened if adding characters.
Passing as a parameter	Passed by reference	Passed by value
String length	strlen(char *)	length(string)
Length of empty string	0 (excludes null terminator)	0 (see `EMPTY`)
Size limit	Available memory based on *alloc* or *malloc*	Approximately 2 MB, after which it may overflow the call stack
String concatenation	strcat()	& or && operator
String copy	strncpy(), strdup()	set newString = oldString
String literals with escape characters	Use backslash character to “escape” unprintable or illegal characters	No direct equivalence. Use string concatenation with Lingo constants (including QUOTE) or *charToNum()*
Formatted strings	printf()	put "String" && *variable*

Event and Error Trapping

Director implements a so-called “event loop” in which it continually waits for events. In Lingo, you need only create handlers, such as *on mouseUp*, that respond to these events, such as *mouseUp*. See Chapter 2, *Events. Messages and Scripts*, in *Lingo in a Nutshell*. There is no need to “subscribe” to any particular

events, as events are dispatched to the appropriate scripts automatically.

Director automatically dispatches events to the correct sprite. For example, you need not attempt to determine which sprites were clicked from a frame script. Instead attach appropriate mouse event handlers directly to the sprites of interest.

Frame scripts that manually check which sprite was clicked are a waste of time and extremely poor style. Avoid the following:

```
on exitFrame
  if the clickOn = 1 then
    -- Do action for sprite 1
  else if the clickOn = 2 then
    -- Do action for sprite 2
  end if
end
```

Instead separate scripts containing *on mouseUp* handlers to the sprites of interest. For example:

```
on mouseUp me
  -- Perform some action for this sprite
end
```

Similarly, you should not attempt to loop within an *on keyDown* handler while waiting for a specific key, nor attempt to manually flush unwanted events. If your *on keyDown* handler does not detect the key of interest, simply loop in a frame script using:

```
on exitFrame
  go the frame
end
```

If you allow the playback head to loop as above, Director will continue to process events. Simply ignore those that are not of interest to you.

Error Trapping and Return Values

Lingo does not have robust error-trapping, nor true interrupts. Errors typically generate an alert dialog box. You can perform your own error checking to prevent this. See the section describing *the alertHook* property in Chapter 3, *Lingo Coding and Debugging Tips*, in *Lingo in a Nutshell*. You can use *the result* to get the return value of the last function in Lingo, as you would use *getLastError()* in C.

Debugging

Director has a debugger (see Chapter 3 in *Lingo in a Nutshell*) but it is not as robust as traditional debugging tools. For one thing, you can change the code at run time, in which case the Debugger may warn you that scripts need to be recompiled! If you close the Debugger window, the Lingo execution continues

and Director regains control. You must stop the Director movie itself to halt Lingo execution and recompile your scripts.

In C programming, the application programmer is at the *top* of the hierarchy. If you set a breakpoint in the C debugger, you can step through your program indefinitely. Even if you call library routines to perform lower-level functions, control always boomerangs back to your routine in the debugger.

By contrast, in Director your Lingo code is sandwiched between the library of Lingo commands (on the bottom) and Director's main event loop (on the top). The Debugger can only debug *Lingo* code. It has no access to the code of Xtras, Director itself, or the low-level Lingo library which are written in C. Because Director's event loop is *above* your Lingo code, it periodically regains control and there is nothing you can do to stop it (short of manually replicating all of Director's event processing in a repeat loop, which you do not want to do).

Suppose you set a breakpoint in an *on mouseDown* handler. At the end of that handler, even if you single-step through your Lingo code, control returns to Director's event processing loop. The only way to regain control is to set another breakpoint in another Lingo handler that you expect to be reached at a later time. Thus, there is no way to follow a given event as it passes through the Lingo messaging hierarchy (from sprite scripts, to frame scripts, to movie scripts, etc.).

Likewise, there is no way to cause an immediate break into whatever Lingo code may be executing at the moment. Director will only break into Lingo at which you have *previously* set a break point. The material in Chapters 1 through 3 of *Lingo in a Nutshell* will help you to set your breakpoints where they are sure to be reached.

You can interrupt execution of a Director movie using `Cmd-.` (Mac) or `Ctrl-.` (Windows), but you aren't afforded the opportunity to enter the Debugger when the movie halts. In fact, there is a good chance that Director's C code, and not the Lingo interpreter, is running at any given time. So there is no way that Director can pass control to the Lingo Debugger at the arbitrary time at which the user halts the movie's execution.

File I/O, Character Streams, and Printing

Most Lingo file input and output is done using the FileIO Xtra, which is included with Director. Other Lingo commands, such as *setPref* can be used to write small text files. Xtras are available to write binary files or communicate with the serial port, etc. Refer to Chapters 13 and 14 in *Lingo in a Nutshell*.

Editable fields automatically accept keystrokes, but you can use an *on keyDown* handler to trap keyboard input explicitly. Refer to Chapter 10, *Keyboard Events*, in *Lingo in a Nutshell*.

Instead of C's formatted *printf* statement, use Lingo's *put* command to print items in the Message window, which behaves as *stdout*. Lingo does not support

"pipes" or a general "redirection" mechanism as is common in Unix and DOS, but you can redirect the output from the Message window to a log file using *the traceLogFile* property.

Operators and Keywords with Different Meanings

The same symbols, operators, or keywords may have entirely different meanings in Lingo versus C, as shown in Table 20-11.

Table 20-11: Symbol and Keyword Differences in Lingo and C

Symbol or Keyword	**C**	**Lingo**
[]	Enclose expression that yield array subscripts	Enclose elements in list declaration. In D7, square brackets are also used to access a list element or a chunk expression within a string.
&	Address operator	String concatenation
&&	Logical *and*	String concatenation with space added
, (comma)	Comma operator used to separate multiple operands on right side of assignment statement.	Separates values in subordinate clause of *case* statement
=	Assignment	Assignment or comparison
*	Multiplication or indirection	Multiplication only
|	Bitwise inclusive-or	Used to define custom menu items with *installMenu*
:	Used with conditional operator ?:	Separates a *property* and its *value* in a property list
-- (two minus signs)	Decrement operator	Comment delimiter
case	Subordinate clause in a *switch* statement	Starts a *case* statement
char	Declares a character variable	Refers to a character in a chunk expression
continue	Continues with next *repeat* loop or *for* loop iteration	Restarts the playback head after a pause (obsolete)
delete	Deletes a pointer	Deletes characters from a chunk expression
do	Part of a *do...while* loop	Executes a command string

exit	Quits program	Exits current handler
goto vs. go to	*goto* jumps to a named section of the current handler	*go to* moves the playback head to a different frame (does not affect Lingo execution).
int or integer	*int* declares an integer variable, or casts a value to (int).	*integer()* coerces data to an integer type
long	*long* declares a long integer	*long* affects the date and time format
RETURN	Returns data from function	Returns data from function or the RETURN constant
short	*short* declares a short integer	*short* affects the date and time format
static	*static* declares a static variable or private function	The *static of member* property controls whether Flash sprites are redrawn.
void	void data type	VOID constant

Lingo Object Oriented Programming Versus C++

Lingo provides an object-oriented model, described in Chapter 12, *Behaviors and Parent Scripts*. Director allows you to mix and match OOP and procedural programming. Table 20-12 compares the terminology for Lingo and C++.

Table 20-12: Lingo vs. C++ Terminology

C++ Term	**Lingo Term**
Class	Parent script
Base class or super class	Ancestor script
Instance	Child Object or child instance
Inherited or derived class (descendant)	A child object with an ancestor
Handler or procedure	Method
Instance variable or member variable	Property variable
Constructor	*new* method
Deconstructor	None. Set variable referencing object to 0 or VOID.
this	*me*

Class Hierarchy and Inheritance

Lingo's class hierarchies are much looser than C++'s. In C++, a class's *base class* is defined in the *constructor*, before the class is even compiled. In Lingo, a child object's *ancestor* property is set at run-time, and can even change dynamically. In Lingo, a child object can only inherit from a single ancestor script (multiple inheritance is not supported). For a child object to inherit additional methods from additional ancestors, its direct ancestor would have to declare its own ancestor, and so on. Note that in Lingo, the term *parent* refers not to a *parent class* or *base class*, but merely to the *parent script* (i.e. class definition) of which the *child object* is an instance. In the simplest case, a single parent script is instantiated to create one or more copies of itself (so-called child objects). Thus a child object is not necessarily a descendent of some base class, and its parent script is not necessary an *inherited* or *derived class*.

The following features of C++ have no equivalent in Lingo:

Abstract Class

Multiple Inheritance

Templates

Encapsulation

It is impossible to *encapsulate* a child object fully. Lingo objects can be manipulated by some list functions, and they are passed by reference, as are Lingo lists. The Lingo *count()* and *getAt()* functions can extract a child object's property variables. These properties can also be read or set by code external to the object using:

```
set the property of childObject = value
```

In D7 notation, you can use:

```
childObject.property = value
```

There is no Lingo equivalent to C++'s `static`, `public`, `private` and `friend` handlers. Methods within a child object are private by default, but any message can be sent to a child object using the *call()* function. It's ancestor can be called using *callAncestor()*. A child can refer to its ancestor's properties using:

```
set the property of the ancestor of me = value
```

Conclusion

Lingo and Director will not impress most C developers immediately (many of whom think scripting languages are for wimps). But as you learn more about Director, you'll find Lingo well-suited to multimedia development. Hopefully this chapter has shortened Director's learning curve, and will help you to tailor your programming style to Director's strengths, while side-stepping its weaknesses. The second biggest obstacle facing new Director developers is that they are not hooked into the larger Director community (the biggest obstacle is your ego). Refer to the resources cited in the *Preface* of *Lingo in a Nutshell*

which are chock-full of very experienced Director developers, many of whom know more about Director than you know about C (if only because there is so much more to know).

You Can Still Call Home

An adept C/C++ programmer can develop their own Xtras to extend Director in outlandish ways. Refer to Chapter 13, *Lingo Scripting Xtras and XObjects*, in *Lingo in a Nutshell* and to Chapter 10, *Using Xtras*, in *Director In a Nutshell*. For details on some of the existing Xtras see the URLs in the *Preface* that point to Macromedia's *Xtras Developer Center*.

If nothing else, *Lingo in a Nutshell* and *Director in a Nutshell* should help you complete a given project in Director, after which you can flee back to C development if you so choose. :-)

As of March 31, 1999, the Director 7 version of the Xtra Developer's Kit (XDK) has not been released. The D6 XDK can be used to develop Xtras for D7 until the new XDK is available. The subject of Xtras development in C is outside the scope of this discussion, but the resources cited at *http://www.zeusprod.com/nutshell/links.html* should get you started.