Bionic Buffalo Tech Note #84

Cats(tm) Programming Language Overview

Abstract

This *Tech Note* briefly describes some of the major features of the Cats™ programming language.

Publisher's information and notes are at the end of this document.

Contents

1. Philosophy and Motivation

The C programming language is not far removed from a portable assembly language. While it cannot take advantage of many features specific to some CPU designs, it allows the most common programming tasks to be programmed close to the metal. Good optimizing compilers often make typical C code almost as efficient as, if not more efficient, than typical assembly language code. Compilers for C are nearly ubiquitous, available for almost all common platforms. Partly for these reasons (but also for historical ones), C is widely used in operating system kernels, embedded systems, and in other applications where added performance and efficiency outweighs the spartan feature level of the language.

Attempts have been made to improve on C, most notably C++. The C++ language has the good features of C, with additional ones, and can be as efficient as C for low-level tasks. It probably would be more widely used for kernel and embedded applications, but for the fact that it arrived later and has taken longer to develop. Unfortunately, C++ compilers are not yet available for many embedded target environments.

Still, at Bionic Buffalo we found ourselves wishing both C and C++ had some additional features, and wanted to add them to the C and C++ languages. The result is Cats, for "*C*, *a*nd *t*hen *s*ome".

We especially liked some features of Perl and C++. For our work, we needed a strongly typed language, and we needed a compiled language. The features we selected are described below.

2. Design Principles

It is impractical to write new compilers for all of the platforms in the world, so we elected a different approach: a pre-compiler. The Cats pre-compiler translates Cats into C, with the added rule that every valid C program must also be a Cats program. Thus, C is passed through the pre-compiler unchanged, and its execution is identical. This also allows Cats and C to be mixed. Since Cats is translated into C, it can be used everywhere C can be used.

The exception we made to the above was in keywords: the pre-compiler recognizes some new keywords. If an existing C application avoids using these new keywords, then it should pass through the Cats pre-compiler unchanged.

Since Cats is translated into C, we could use Cats in house for development, then hand C code to our customers, without having them suffer directly through the development of Cats. Also, we didn't want customers with source code to depend on a tool they didn't have. To make this easier, we made sure that

the generated C code was amply commented by the Cats pre-compiler, so it could be maintained as C code if necessary.

Although new types are defined, the implementation is left to the pre-compiler, and is meant to be opaque to the Cats programmer. For example, it should not be assumed that `string` is implemented as `char *`. In fact, `string` is implemented as a certain structure, but the pre-compiler may change that implementation in some future revision. Operators and functions are provided to access features that are opaque but might be necessary to use in an application.

The primary goal was to augment the C language, while, if possible, allowing the same pre-compiler to be used to augment C++. The C++ work lags far behind the C work.

3. Additions to the Language

3.1. New Keywords and Syntax

The following keywords are added to C and to C++ by the Cats programming language:

```
any
bag
bit
count
foreach
implement
interface
length_of
list
map
object
opaque
plex
regex
sequence
set
sparse
string
thread
type_of
typecode
vacancy_of
value_of
wstring
xchar
xstring
```

The following sections describe how these new keywords are used.

3.2. New Types and Their Use

`string` and **`wstring`** - These are strings of `char` and `wchar_t`. It should *not* be assumed that they are implemented as `char *` (or `wchar_t *`), or that they are `NUL`-terminated. Variables of these types can be concatenated using the `+` operator. In a mixed expression, `char *` will be promoted to `string`, `wchar_t *` will be promoted to `wstring`, and `string` will be promoted to `wstring` as necessary. For example:

```
char * ptr_to_char ;
wstring widestring ;
```

```
ptr_to_char = ( char * ) “fine “ ;
widestring = “a ” + ptr_to_char + L”pickle” ; // “a fine pickle”
```

The `value_of` operator can be used to extract the buffer pointer (`char *` or `wchar_t *`) of variables of these types. A subscript can be used to reference a particular character of the string.

`xchar` represents an ISO 10646 character. (For almost all purposes, this is a Unicode character.) The generated C type is opaque to the application, but it can be assigned to a numeric type to extract the numeric value of the character. Unlike `wchar_t`, a variable of type `xchar` will always be able to contain any ISO 10646 character.

`xstring` is a string of ISO 10646 characters. (Practically, a Unicode string.) It can be mixed with `char *`, `wchar_t *`, `string`, and `wstring` types in expressions; each type will be promoted to the next type as needed.

`list` is a linked list. A variable of this type is declared as `list (X)`, where `X` is the type of element in the list. Lists can be concatenated to form bigger lists (`list1 = list2 + list3`).

Elements can be prepended to the beginning (`elem1 + list1`) or appended to the end (`list1 + elem1`). A subscript can be used to reference a specific element (`list1 [ n ]`). Negative subscripts cause the list elements to be referenced from the end, so `list1 [-1]` refers to the last element.

The `length_of` operator returns the number of elements in the list. The code will not give up the address of a list element: you cannot write `&(list1[n])`.

Using a subscript beyond the bounds of the list simply causes the reference to wrap around. In other words, if `n>length_of list1`, then `list1[n]` is equivalent to `list1[n mod length_of list1]`. If there are no elements in the list, then any reference to an element is equivalent to a reference to a block of `NUL`-filled memory.

When a subscript is followed by an exclamation mark, the referenced element is deleted from the list. For example, `elem1=list1[-1!]` deletes the last element of the list, assigning its value to `elem1`.

`sequence` is almost the same as list, except for the implementation. Whereas `list` is implemented as a linked list, `sequence` is implemented as a flexible vector of the elements. Otherwise, `sequence`s are the same as `list`s. A sequence is roughly equivalent to an array with flexible bounds.

If a `sequence` must be expanded, then the entire sequence buffer will be re-allocated, and the old buffer copied to the new buffer. If an element is prepended to the beginning, then all the other values must be pushed up to make room. If an element is deleted from the middle, then the other elements above it must be moved downward to fill in the space. These can be expensive. The runtime will attempt to pre-allocate a buffer bigger than necessary, so reallocation is minimized. Furthermore, the buffer will be allocated in segments, and indexed, so the maximum disruption will be to a segment rather than to the

entire buffer. On the other hand, adding elements to a `list` always requires memory allocation. Some applications will work better with `sequence`s, others will be more efficient with `list`s. The best choice will depend largely on how stable the number of elements will be, whether insertions or deletions are made to or from the middle, and how big each element is.

`plex` is an array with variable bounds. The minimum or maximum subscript values may be positive or negative. A `plex` is implemented similarly to a `sequence`, and behaves the same except that negative subscripts do not reference elements from the end.

`sparse sequence` and **`sparse plex`** allow missing elements. In other words, there may be no element corresponding to a subscript. As with ordinary sequence or plex types, deleting an element using the exclamation operator (as in `seq1[n!]`) causes the remaining elements to move up or down to fill the space. With `sparse sequence` or `plex`, using a double exclamation mark (`seq1[n!!]`) causes the subscript to become invalid, even though it is in the middle of a valid sequence of subscripts.

The `vacancy_of` operator refers to the value of the `sparse sequence` or `plex` to be used when reference is made to a nonexistent element.

`bit` represents a single bit, while **`bit(n)`** represents an array of `n` bits. These types can be used in arrays, sequences, `list`s, and `plex`es. Individual bits are referenced using subscripts. Unlike with the bit field notation (such as `struct X { int a:5; }`) available with C, the `bit` type is guaranteed to be packed as tightly as possible, and does not have to appear within a structure. Furthermore, Cats implementations of `sequence` and `plex` can optimize access to large `bit` types.

`typecode` is an opaque type which describes the type of a variable. There are various operations on `typecode` which return values allowing an application to know the variable's type and parameters. For some types (such as `int` or `unsigned short`) there are no type parameters. For other types, such as arrays and structures, additional parameters are needed for length, member types and so on.

`any` is a type which can contain any type. The `type_of` operator returns the `typecode`, and the `value_of` operator refers to the value. Assigning to an `any` sets the type and value. Assigning to `value_of any1` is equivalent to assigning to a fixed type.

`set` is an unordered collection without duplicates. A `set` containing values of type `X` is declared `set(X)`. Sets are manipulated the same way as lists and sequences, except that attempting to add an element which duplicates an existing element is ignored.

`bag` is a set which allows duplicates.

`map` is an association between a key and content. It is declared by `map(X,Y)`, where `X` and `Y` are the key and content type. It is similar to an associative array or Perl hash. The content is referenced by using the key value as a subscript. Elements are added by assignment to a subscripted reference (as `map1[key1]=content1`), and removed using the exclamation mark notation (as `map1[key1!]`).

regex is the type of a compiled compiled regular expression. The implementation of a regex is opaque. A string may be cast to a regex, and *vice versa*.

3.3. Regular Expressions and String Operations

In addition to understanding the + operator for concatenation, Cats also understands some other string operations.

3.3.1. Pattern Matching

The expression

```
string1 $* regex1 $: options1
```

where all three arguments are strings, returns a list of matched strings. This is similar to the Perl expression $string1 =~ m/($regex1)/<*options1*>. The strings within the list are equivalent to $1, $2, and so on, in Perl. If there are no matches, or if there are no capturing parentheses, then an empty list is returned. After the operation, the predefined variable count contains the number of matches. For example:

```
string a = "wildebeest feast" ;
list (string) b ;
b = a $* "(e[ae]st)" $: "g" ;
                          // b contains "eest" and "east"
```

The match/assignment operator $*= is defined, in the same was as are +=, -=, and so on.

3.3.2. Substitution

The expression

```
string1 $% pattern1 $/ replacement1 $: options
```

returns the value of string1 after the substitution. After execution, the predefined variable count contains the number of substitutions.

The Perl expression $string1 =~ s/$pattern1/$replacement1/<*options1*> alters the value of $string1. By contrast, the equivalent Cats expression returns the altered value, without affecting the original value of the string1 argument. A Cats program can use the substitute/assignment operator $%= (in the same way that += and -= are used) to change the value of the string1 argument.

3.3.3. Transliteration

The expression

```
string1 $- search1 $+ replace1 $: options1
```

causes transliteration: if a character from `string1` matches a character from `search1`, then it is replaced by the corresponding character from `replace1`. The number of substituted characters is returned in the predefined variable `count`.

This is equivalent to the Perl expression `$string1 =~ tr/$search1/$replace1/<options1>`, except that the Cats expression returns the transliterated result rather than the number of substitutions.

3.3.4. Splitting

The expression

```
string1 $^ delimiter1 $: options1
```

returns a list of substrings created by splitting `string1` using a pattern `delimiter1`.

3.3.5. Compiling Regular Expressions

A regular expression is compiled from a string by simple assignment:

```
regex re1 ;
re1 = "[a-zA-Z0-9]" ;    // compiles the string into a regex
```

The reverse is also possible. Either may be cast to another within an expression. (That is, one can write `a = (regex) b`, or `b = (string) c`.)

3.4. Other New Expression and Syntax Features

3.4.1. Ranges

The range operator (`..`) allows reference to multiple elements of an array, `sequence`, `plex`, or `map` at one time. For example, the notation `array1[4..6]` refers to an array consisting of the 5th, 6th, and 7th elements of `array1`. When ranges are used with `map`s, the sort order for the keys is a simple binary comparison of the key values, which may have unexpected results in some cases. When assigning to a range, the effect is as if only the referenced elements were assigned. For example,

`string1[1..2]="ab"` assigns to the 2nd and 3rd characters of `string1`, leaving the other characters of `string1` unaffected.

Disjoint elements of an array, `sequence`, `plex`, or `map` may be referenced by separating subscripts using semicolons. For instance, `array1[3;5..6]` represents the 4th, 6th, and 7th elements of `array1`.

3.4.2. Anonymous Lists

An anonymous `list` may be written by enclosing its elements, separated by semicolons, within square brackets (`[` and `]`). The range operator may be used with integers. For example, the list

```
[ 0; 2..3 ; 5 ]
```

contains the elements `0`, `2`, `3`, and `5`.

The square brackets used for subscripting are a special use of an anonymous `list`.

An anonymous `list` of lvalues may be used as an lvalue. For example,

```
list ( unsigned ) a = [ 2 ; 3 ] ; // initialize a
int j, k ;
[ a[1] ; a[0] ] = a ;          // swaps a[0], a[1]
[ j ; k ] = a ;                // now, j = 3 and k = 2
```

This allows extraction of the individual elements of a `list`.

3.4.3. Reverse

The `%%` operator reverses the sequence of elements in a `string`, `array`, `list`, `plex`, or `sequence`. It does *not* affect the operand, only the way the operand is viewed in an expression.

For example,

```
string a = "abc", b ;
b = %% a ;          // a is unchanged, b is "cba"
```

3.4.4. Sparse Increment and Decrement

When accessing a `map`, or a `sparse sequence` or `plex`, it is sometimes convenient to know the next lower or higher key or subscript. This can be done using the `++` or `--` operators in the context of the subscript. For example, `map1[key1++]` will set `key1` to the next higher key value after making the reference, and `map1[--key1]` will "decrement" the key before making the reference.

3.4.5. Iteration Over Lists

The `foreach` statement may be used to iterate over all of the elements in a `list`, `sequence`, `array`, `plex`, or `string`. For example,

```
list (string) a ;
foreach b ( a )
     { // some code goes here } ;
```

The code in the loop is executed as many times as there are elements in the `list`. The value of the variable is set, in turn, to the values of every element of the `list`.

The `list` may be anonymous.

3.5. Other Features

In addition to the above, Cats includes support for exceptions, threads, objects, and other features. The compiler also will generate additional code for debugging support. For complete details, the reader is referred to the *Cats™ Language Reference Manual*.

4. Garbage Collection

When the `counted` keyword is used for a type, then pointers to variables of the type are counted. When no pointers to an entity remain, then the entity is destroyed automatically.

This is not a perfect solution: it can be defeated by assigning a `counted` pointer value to a `void*` pointer. However, it can considerably reduce the number of inadvertent orphaned allocations.

5. Generated C Code

The Cats pre-compiler accepts Cats source code emitted by the C pre-processor, and emits C code compliant with the C99 standard.

Ordinary C code is passed through the pre-compiler unchanged. Cats statements are replaced by ordinary C statements implementing the semantics of the Cats statements. The original Cats statements may be passed through as comments, and may be supplemented by additional commentary to aid in understanding the generated code.

Some Cats statements can be replaced by C code at the places in the program where the Cats statements were found. For example, type declarations without any necessary initialization translate directly at the point of declaration.

Other Cats statements require the generation of code at points removed from the same Cats statements. For examples:

- Declaration in a block of a structure which requires memory allocation may also require the generation of code at each exit from the block to release the allocated memory.
- Expressions which require the use of temporary variables may require the declaration of those temporary variables at the appropriate places.

Some debugging features will cause the generation of additional code for C programs which do not contain any Cats statements. For instance, a traceback message may be generated upon program failure, providing the names of procedures called up to the point of failure; such a traceback message requires that each program push its name onto the traceback stack as it is called, and pull it off upon exit.

Aided by the values of run-time switches, the pre-compiler will attempt to match the superficial coding style of the original programmer when generating code. For example, the pre-compiler will attempt to use the same indentation and spacing as used in the source code.

6. Subset Pre-Compiler

To enable a prospective user to evaluate Cats, a free, subset pre-compiler, called Kitten™, is being developed. The subset pre-compiler supports strings, but not the other data types. Kitten also supports regular expressions and the related syntax of Cats.

7. Future Directions

Bionic Buffalo is considering various enhancements to Cats and to the pre-compiler. Among these are:

- Support for additional string quoting mechanisms, similar to `qw` and other Perl constructs. Because the C preprocessor becomes confused by these quoting mechanisms, this requires processing the code before it is seen by the C pre-processor. (It may be most practical that the Cats pre-compiler will replace the C pre-processor.)
- Association of variables with files or with URLs, so that their values are persistent, and so that very large data structures can be accessed in small amounts of memory. This might be viewed as a specialized kind of virtual memory.
- Integration with underlying file and directory system. Under consideration are file and directory data types. It would be useful to enable Cats to do things with files and directories, now normally done with shell scripts.
- Support for literate programming: documentation and other information directly embedded in the source code, or to be produced from the source code.

- Facilities to support configuration management, testing, and additional debugging capabilities.

Possible changes to licensing, including perhaps open-sourcing the pre-compiler, are also under consideration.

Feedback and suggestions (to *support@tatanka.com*) are welcome.

Bibliography and References

Bionic Buffalo, *Cats™ Programming Language Reference Manual*

Bionic Buffalo, *Cats™ Pre-Compiler Manual (Zambia)*

Bionic Buffalo, *Cats™ Run Time Library Reference Manual (Malawi)*

Bionic Buffalo, *Kitten™ Pre-Compiler Manual (Mozambique)*

Publisher's Information and Notes

Publisher. Bionic Buffalo Corporation, 502 North Division Street, Carson City, Nevada 89703, USA.

Current Version. The current version of this document may be found at *http://www.tatanka.com/doc/technote/tn0084.pdf* (for PDF) and *http://www.tatanka.com/doc/technote/tn0084.html* (for HTML).

Other *Tech Notes*. For available *Tech Notes*, visit the Bionic Buffalo web site at *http://www.tatanka.com/doc/technote/index.htm*, or e-mail *query@tatanka.com*. Most Bionic Buffalo *Tech Notes* are available in both HTML and PDF form.

Key. PGP/GnuPG key fingerprint: `a836 e7b0 24ad 3259 7c38 b384 8804 5520 2c74 1e5a`.

Trademarks. *Tatanka*, *Bionic Buffalo*, *Cats*, and *Kitten* are trademarks of Bionic Buffalo Corporation. Other trademarks may appear within this document, and, if so, belong to their owners.
