Unimal 2.0

Application note 1

Managing unstructured data layouts from legacy code

Documentation revision 2.00

Techniques:
Relations between strings and names in Unimal

MacroExpressions
http://www.macroexpressions.com

Table of contents

Managing unstructured data layouts from legacy code

This somewhat weird problem is hard to make up; it appeared in a real application. The problem has to do with record layouts in the files produced by an older application. As a live example, consider a flat ASCII file of records, where each record can be mapped to the following C structure:

```
struct myrecord {
      char FirstName1[10];
      char StreetAddr0[30];
      char LastName2[16];
      char StateZip2[40];

      char StreetAddr3[30];
      char LastName1[16];
      char FirstName3[10];
      char StateZip0[40];

      char StreetAddr2[30];
      char FirstName0[10];
      char LastName3[16];
      char StateZip3[40];

      char StateZip1[40];
      char FirstName2[10];
      char StreetAddr1[30];
      char LastName0[16];
} OldRecord;
```

In this example, a record contains information on four items each consisting of the first and last name, street address and state/ZIP. However, the layout of the record is such that all parts of an item are found in a strange order and intermixed with parts of other items.

There must have been a good reason why such a layout was implemented. However, now that the secret of the ancient craft is lost, any processing would rather use an array of four structures, each representing one item, like this:

```
struct item_record {
      char *FirstName;
      char *LastName;
      char *StreetAddr;
      char *StateZip;
} NewRecord[4];
```

To make use of the `NewRecord` array, we need to initialize all the pointers, like

```
NewRecord[0].FirstName=OldRecord.FirstName0;
.................
NewRecord[3].StateZip=OldRecord.StateZip3;
```

Doing so by hand is tedious and invites all kinds of errors. However, generating the same code with Unimal is very straightforward:

```
#MP Expand Old2New(#@FirstName#, 4)
#MP Expand Old2New(#@LastName#, 4)
#MP Expand Old2New(#@StreetAddr#, 4)
#MP Expand Old2New(#@StateZip#, 4)
```

The macro `Old2New` simply initializes the element named in the string argument for the number of items given in the second (numeric) argument:

```
#MP Macro Old2New ;(field_string, item_number)
 #MP For Count=0, #2#-1
   NewRecord[#mp%dCount].#mp%s#1# = OldRecord.#mp%s#1##mp%dCount;
 #MP Endfor
#MP Endm
```

All this simple macro does is rendering the translation line the specified number of times. There the `NewRecord` array offset is Count (rendered with `#mp%d` as decimal), which is also a tag of an item in the `OldRecord`. In addition, the `NewRecord` item name, which also is an item name in the `OldRecord` (stripped of its tag), is rendered with `#mp%s` as a string.

That's all we needed to make the translation maintainable. There is a file, old2new.u, in the folder Samples\AppNotes\1, which implements exactly this. You may want to run it to see the output.

Using simple names instead of strings

Notice that all string literals in the example above are single words not starting with a letter. (It is a "duh!" because they stand for identifiers in the C language – an underscore counts as a letter.) But that means that they can be Unimal simple names.

So, if you think that the following text:

```
#MP Expand Old2New(FirstName, 4)
#MP Expand Old2New(LastName, 4)
#MP Expand Old2New(StreetAddr, 4)
#MP Expand Old2New(StateZip, 4)
```

– looks simpler, or at least less cluttered, here is an opportunity. Let's redefine `Old2New` to use *names* instead of *strings*.

If you think of it, all we used with a string macro argument for is to render it with a %s format. But the same effect can be achieved by rendering a name argument with the %n format! Here is the modified macro:

```
#MP Macro Old2New ;(field_name, item_number)
 #MP For Count=0, #2#-1
   NewRecord[#mp%dCount].#mp%n#1# = OldRecord.#mp%n#1##mp%dCount;
 #MP Endfor
#MP Endm
```

The file old2new1.u in Samples\AppNotes\1 illustrates this solution; you may want to try it and see that the output is the same as before.

How close are names and strings?

This simple example also illustrates an important fact: A name can be passed as a macro argument before it is defined (i.e., assigned a value). Actually, in our example we never even needed to define the names we used – that's because we never relied on their values.

Interestingly enough, there is a close relation between names and strings in Unimal:

If S is a string then %sS is a (composite) name containing the same characters in the same order as S. In particular, the renderings in the target language interface, #mp%sS and #mp%n%S produce the same results. Whether or not %sS can be referenced by a simple name depends on the content of S: it must be a single word.

Conversely, if N is a name, then {N} is a string containing the same characters in the same order as N.

This close relation between strings and names allows to do odd things. For instance, the following:
#MP Setstr S=""
#MP %S=5
assigns a numeric value (5) to a macro parameter with an empty name. The empty name is a valid composite name but since it is not a word, it cannot be used as a simple name literally:
#MP =5
is a syntax error. However, like any other composite name, it can be a macro argument. The file `oddities.u` illustrates this; please, take a look.