Simplifying your Coding Life with Custom Types

Göran Tornquist

Introduction

In my other paper created for this Lasso Summit, I have described the benefits gained when using custom types together with access to database records. The focus on this paper is to give the foundation for discussions about general benefits about custom types and how they could be implemented.

I have defined three custom types to introduce you to the benefits of custom types. They will be briefly described by the following subtitles.

- What is the purpose of the custom type?
- Member tags.
- Examples of usage.

Which is the Faster? Custom Types or Custom Tags?

That is a question that has both a simple and a more elaborate answer. The simple, and generally true answer is that custom tags are faster than custom types.

Everything comes with a price

Custom types come with two extra costs: extra development time to handle the general case instead of the specific case, and inefficiency due to lack of optimization.

A custom type is a definition of a "thing" that has the attributes and the knowledge to manipulate those attributes. Normally, optimizations are not present. They would require information outside of the scope for the custom type. That would break the rule that says that custom types (or classes as in other OOP languages) should be self-contained.

There's more inefficiency as well. Lasso has to handle the encapsulation of the custom type member variables and member tags. This requires some overhead, as every object – a created instance of the custom type – does need a general preparation. To minimize this, there is the `-prototype` keyword that can be used together with the definition of a custom type. When doing this you tell Lassoscript that there is no code that needs to be executed at the time of the creation of the object. In other words, there is no outside dependency that has to be accounted for. Using `-prototype`, simply makes Lassoscript copy a ready-made template, which makes it almost as efficient as the native types that are built-in into Lassoscript.

What is the Definition of Fastest Code? Quick and Buggy Code or Slower and Correct Code?

If you feel challenged by this subtitle, please understand that I do not prefer OOP at all times. There are a lot of cases where it does not apply for reasons of efficiency, complexity and interchange with other languages – just to name a few. Now that you've read this disclaimer, please proceed.

The purpose of OOP is to make a simplified and general abstraction of a certain entity that is found in a solution. Almost every thing in this world appears more complex when we look a bit closer at it. Custom types give you the possibility to create a complex operation that will look very simple from the outside.

The question above is a rhetoric question with the assumption that it is easier to create bug free code with custom types. Well-defined behavior can be tested through unit testing and therefore we can predict the most of the results of the behavior. Also, since the code only relies on outside information passed on through the member tags, we know that the consistency of the object is preserved at all times.

Are Custom Types Cost Efficient?

The answer of the question above depends on the resources and the usage of the solution in a whole. Which boundaries do you operate within? The two main factors are time and cost – and both results in money in the end.

- If you have a lot of time before the solution needs to be there, then you can afford to create optimized and specific code.

- If you can afford multiple resources working on the solution, then you will have a high performance solution. Resources is regarded to be many developers working on the problem, or many servers co-operating to deliver the solution.

Time and cost are often restricted by budgets or deadlines, and can be regarded as absolute in the specific case. But there is actually one more factor that gives us the boundaries for the solution. The quality of the solution as whole must not be negotiable, however completely bug-free code is impossible and therefore there's no such thing as an absolute value of the quality factor.

The main cost benefits from custom types can be found in code reuse and less quantity of bugs. Full code reuse takes place when you can create a custom type for one solution, and you use it in another solution without any changes. Whether you will be able to take advantage of full code reuse or not, depends on the balance between the depth of investigation of the abstraction and the cost (in time and resources) you that can afford.

Often there will be room for extension of the custom type, which will affect and benefit both the old and the new solution. Obviously that demands that the extension does not violate the boundaries of the custom type abstraction. If it does, then we have a back compability problem.

C_URL – a Custom Type for handling URL's

You find the custom type C_URL in the file "ctypes.inc" that comes with this paper.

What is the purpose of the custom type?

To deliver correct handling and evaluation of URL's (or URI's) as defined by W3C. Since this is a simple example, the full purpose has not been fulfilled.

Member Tags

You use the custom type trough one of the following member tags. All getters will return a valid value if the URL as a whole is considered valid.

Member Tag	Purpose
getURL	Guaranteed to either deliver a working URL that is consistent with the definition of a URL and the usage depending on the supported protocol, or an empty string if the url is not valid.
setURL: 'url string'	Accepts a string that defines the URL, and stores it within the object.
getProtocol	Returns the protocol part of the URL.
setProtocol:'protocol string'	Sets the protocol part of the URL with the given protocol string.
getHost	Returns the host part of the URL.
setHost: 'host string'	Sets the host part of the URL with the given host address.
getPath	Returns the path part of the URL.
setPath	Sets the path part of the URL.

Examples of Usage

example a

```
var: 'homepage' = (C_URL: 'http://www.cortland.se/summit/');
'a: ';
$homepage->getURL;
'<br />';
//example b
var: 'html' = '<a href="' + $homepage->getURL + '">';
'b: ';
if: $homepage->getProtocol == 'mailto:';
  $html += 'Send a mail';
else;
  $html += 'Click here';
/if;
#html += '</a>';
#html;
'<br />';
//example c
'c: ';
$homepage->setProtocol: 'ftp:';
$homepage->setHost: 'ftp.cortland.se';
$homepage->getURL;
'<br />';
```

C_String – an Extension of the Native String Type

You find the custom type C_String in the file "ctypes.inc" that comes with this paper.

What is the purpose of the custom type?

Two purposes:

1. To prove the possibility of using strings that has been extended by use of inheritance.
2. To simplify code that manipulates the string – and thereby avoid bugs.

Comment: When the [String_CountFields] and [String_GetField] tags where deprecated, I was forced to change my solutions in quite a few places to accommodate for the new and more efficient [String->Split]. But as I was changing those, most of the time, there was not time and resources available to change the structure of the code. Without a structure change, the benefits of performance were missing. Also I found myself checking for just one field to be present, and

then continue with some processing. The code became more complicated and less readable, which is a threat to future development and handling of bugs.

So, here they are again. Built with [String->Split], with the promise of low performance, but simpler looking code.

Member Tags

You use the custom type trough one of the following member tags.

Member Tag	Purpose
left: 'length'	Returns a maximum of 'length' characters from the left side of the string.
right: 'length'	Returns a maximum of 'length' characters from the right side of the string.
countFields: 'delimiter'	Returns the number of string fields that are divided by the given delimiter.
getField: 'delimiter', 'position'	Returns the string field at the given position, delimited by the given delimiter. If not found, it will return null.

Examples of Usage

```
var: 'name' = (C_String: 'Göran Törnquist');
//example a
'a: ';
$name->(left: 5);  //returns 'Göran'
'<br />';
//example b
'b: ';
$name->(left: 0);  //returns 'Göran'
'<br />';
//example c
'c: ';
$name->(right: 5);  //returns 'quist'
'<br />';
//example d
'd: ';
$name->(right: 256); //returns 'Göran Törnquist'
'<br />';
//example e
'e: ';
$name->(right: 0); //returns the empty string
'<br />';
//example f
'f: ';
var: 'address' = (C_String: 'ftp://ftp.cortland.se/summit/');
$address->(countFields: '//'); //returns 2
'<br />';
//example g
'g: ';
$address->(getField: '//', 2); //returns 'ftp.cortland.se/summit/'
'<br />';
```

C_Integer – an Replacement of the Native Integer Type

You find the custom type C_Integer in the file "ctypes.inc" that comes with this paper.

What is the purpose of the custom type?

Two purposes:

1. To prove the possibility of replacing the native integer, in certain cases, yet be able to fully calculate without implicit conversions.
2. To make formats, that has been set by `C_Integer->setFormat`, not be reset everytime we perform a calculation or assign a value to the integer.

Comment: There was no way I could implement this using inheritance from the native integer type. The reason for this is possibly the optimizations done in the implementation of the member tags of the integer type.

Member Tags

There are no special member tags that differs from the normal integer. The main feature of this custom type is that it is not any different from an integer when it comes to calculation or the alike. However, once a format has been set using C_Integer->setFormat, it will stick. No matter the assignments being done to the variable that contains the object.

Examples of Usage: The native integer behavior

```
var: 'salary' = 1000;
$salary->(setFormat: -groupChar=' ');
//example a
'a: ' + $salary + '<br />';
//example b
$salary += 5;
'b: ';
$salary + '<br />';
//example c
$salary = 2000;
'c: ';
$salary + '<br />';
```

Examples of Usage: The C_Integer behavior

```
var: 'salary' = (C_Integer: 1000);
$salary->(setFormat: -groupChar=' ');
//example a
'a: ' + $salary + '<br />';
//example b
$salary += 5;
'b: ';
$salary + '<br />';
//example c
$salary = 2000;
'c: ';
$salary + '<br />';
```

Epilogue

I expanded my own knowledge while researching for this paper. I had a clear and distinct idea what I wanted to prove with the paper and the presentation. The last custom type, C_Integer, took quite some time to realize and to work 100% the way I wanted it to. The path to fully understanding the inheritance and overloading of callback member tags and symbols was a long and troublesome one.

If I would be lazy, I'd blame either the implementation of Lasso, or the documentation for this, but I don't. When it comes to the inner workings of inheritance and the full operations of a cascading member tag, it is not an easy subject.

One of the reasons for my troubles was the implicit conversion of data, which made me suffer from a infinite recursion until I finally understood how it works.

The first two examples are actually quite simple. The advanced part in them lies in understanding how the callback member tags are working.

The third example is more of a subject for the fearless. The reason I developed that custom type was that I was tired of keeping track of whether formatting was done or not. With this custom type in place, I can rely on the formatting being in place when I'm converting to strings for presentation.

Happy ctyping

/Göran