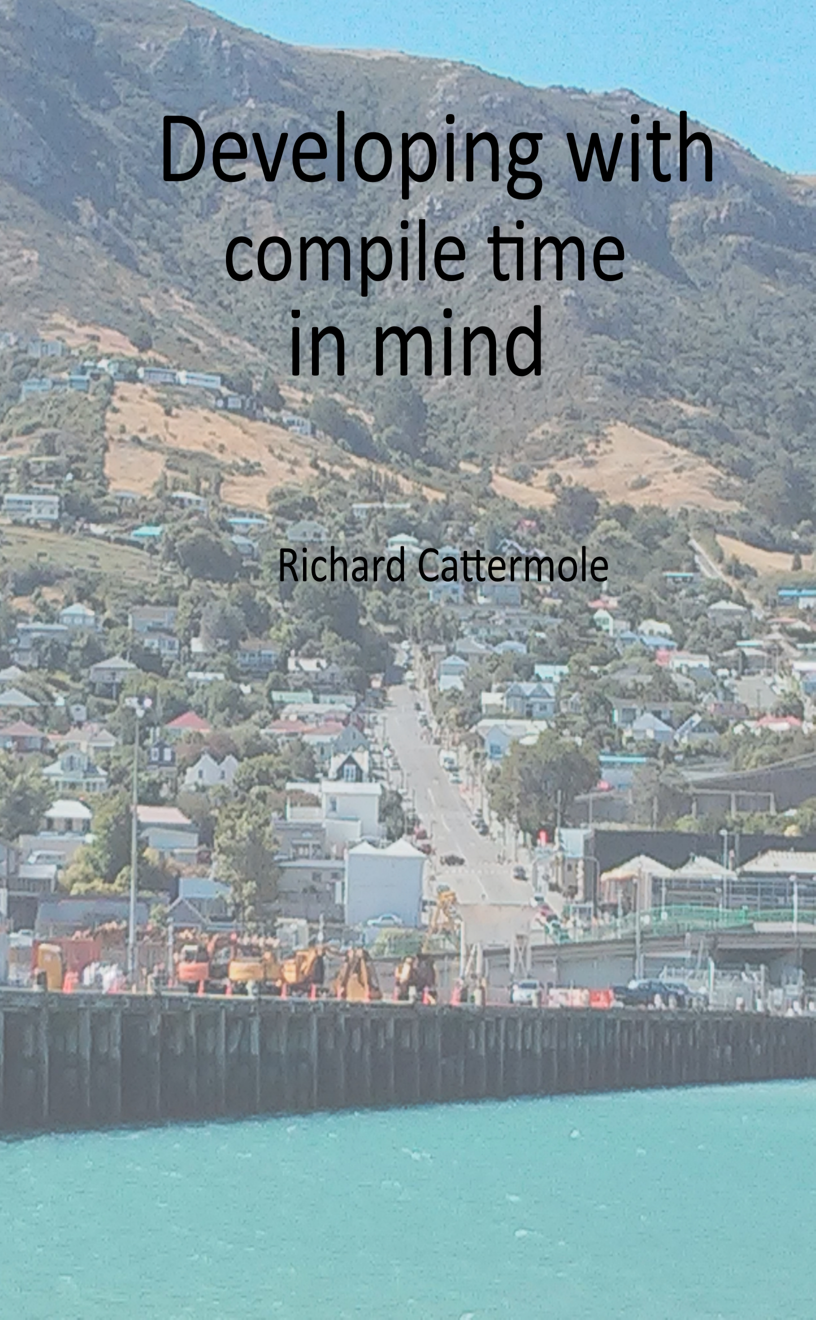

Developing with compile time in mind

Richard Cattermole

Developing with compile time in mind

richard cattermole

This book is for sale at
http://leanpub.com/ctfe

This version was published on 2015-10-12

Tweet This Book!

Please help richard cattermole by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ThinkCTFE.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#ThinkCTFE

Also By richard cattermole

Contents

Introduction

Developing software is an amazing process that can take many different forms. In the case of this book we look towards a little known and used concept. Compile Time Function Execution or Evaluation for those inclined, is the ability to execute code during the compilation phase of a program. This opens up many different avenues for automation of code generation. But it is a double edged sward. The time it takes during compilation is heavily dependent upon what the compiler must do.

When reading this book, have in mind that very few languages actually support CTFE currently. Those that do generally do not support it to the extent that is expected for this book. Because of this and the experiance with the D programming language. It has been chosen by the author to use D as the base language refered to within the book.

Exploring language support

Language support for CTFE is rather varied. From the most utter basic of constant expansion such as parsing of a number literal. To the more complex of having a GUI toolkit running.

Because of how varied CTFE support can be, it must be classified. The LISP family languages are a great starting point in everything from simple to complex designs. The D programing language on the other hand, can be more familiar because of its C family origination. Unlike the C/C++ where the macro preprocessor was used, D does not have this. However in some ways the macro pre-processor in C/C++ could be considered a form of CTFE.

The D programming language

To fully grasp the usage of CTFE in a codebase. Let's start off with a language that has a semi decent support. For this the D programming language will be used. D has good CTFE support but it does have limitations which are intentional.

D has meta-programming support which has been described as compile time arguments. A particularly unique language feature D has is known as a mixin template. A mixin template is a template that instead of creates a new type, can be effectively be thought of as output AST in a given context[1].

A mixin template should not be confused with a normal template. Where a normal template effectively creates a new symbol uniquely to the arguments given[2].

[1]http://dlang.org/template-mixin.html

[2]http://dlang.org/template.html

To further make use of CTFE code, it can be useful to take input in some form or another. That does not rely on hard coding into the file the values. There are two known ways for this, string imports and using a tool such as Bin2D[ˆBin2DGithub].

String imports have the syntax of `enum string thefile = import("file");`. The search path for the file must be provided to the compiler via the `-J` flag or for dub via `stringImportPaths` property.

Bin2D is a very useful tool in that it can generate a file that contains many directories or files as byte arrays. Alternatively it can also export at runtime on request. However because of it being an external tool it does require a preprocess action which can be slower.

Developing CTFE'able code

To develop D code compatible with compile time, there is one main restriction. All required information to execute must be passed in. There is no global data in any form including static variables within a function.

As a result of this restriction the `pure` attribute is in heavy use by the author in his code bases. The `pure` attribute reflects the same restrictions that CTFE comprises of. No access to global data. Most of the time it would also be wise to add `@safe`. Where by removing the direct use of pointers.

An exception to the rule of no global data is constants. A constant such as enum is well known by the compiler, as such it can be utilized. However the opposite of this rule is no external code such as extern(C) functions. An example of a CTFE'able function would be factorial.

Factorial function callable at compile time.

```
1  size_t factorial(size_t n) pure {
2          assert(n < 11);
3  
4          if (n == 0)
5                  return 1;
6          else
7                  return n * factorial(n - 1);
8  }
9  
10 static assert(factorial(5) == 120);
```

As shown by the code sample for a factorial callable at compile time it must execute and output the value of 120 during compilation. We know that it must by the static assert statement on line 10.

The pure attribute is listed on the right of the statement declaration before the opening bracket. For those unfamiliar with D this has the same effect as it being of the left. However contested. Some consider it good practice for attributes on the right to refer to the function and the left return type. Further, size_t is used as the data type. This is an unsigned integer dependent upon the word size of the resulting binary. Such as for 32bit, uint.

During optimised targets (`-release`) assert at compile time (not static version) should also be assumed to work and work like a static assert.

The declaration of `size_t` is an alias within two version statements. One for x86 the other for x86_64. Lastly, there is two different types of asserts in D. First the normal runtime based assert and secondly the compile time static assert. The static assert is a declaration at compile time that must be true to compile. Whereas during a debug build an assert will throw an exception if it is not true. This is useful for contract based programming.

However this is not the only type of static statement, there is another: Static if! Static if is just like a regularly if statement except it can conditionally include code based upon the statement at runtime. This is comparable to the #if or #ifdef macro in C/C++. However do note, this is part of the language and not part of a preprocessor like in C/C++ [3].

Going forward it is necessary to discern the difference between runnable at runtime and not. Not all code that is written with CTFE in mind should be ever ran during runtime. This produces three groups of code. 1. Runnable during runtime 2. Runnable during compile time 3. Runnable during runtime and compile time

The first is the most obvious, in most languages you write for this predominately. If manipulation of types is required, runtime based reflection is used. In D however this is not possible comparatively without explicit compile time knowledge and expectation of this. Second, to execute at compile time is mostly what this book is about. However it would be useless if it couldn't produce code to be executed during runtime. This is the third group.

Third, to execute at both compile time and runtime. To execute during runtime, information is passed in during compilation. While this may include the most basic generic like

[3] http://www.cplusplus.com/doc/tutorial/preprocessor/

functionality found in Java. For all all intents and purposes it is not for this book. That is categories under the first, runtime based. This is because while it does change code generation, it does not produce code specifically based upon the input. Code that uses this functionality include ranges and generics for e.g. all three string types (string, dstring and wstring).

Lisp family of languages

The LISP family languages cover a large variety. Common-LISP, Dylan and Converge are great examples. They each have some form of support for CTFE. They base their implementation upon macros. Common-LISP is the most basic of all the support. The macro support evaluating new code based upon the call. But expands out in a forced inline. While this is fairly powerful, Dylan provides much more access to the compiler. Dylan's macros are meant for direct extension to the language for both statements, definitions and operators. What is unique about Dylan is many features that would be considered on a type is moved into macros. Lastly in Converge, CTFE is thought of as compile-time metaprogramming. This is different in other LISP languages primarily in that it gives full access to the AST at that macros entry point. This is by using a "CEI" object. This is considered the main method at CTFE to interact with the compiler.

Different Types

Implementation of any functionality can vary widely. In the case of CTFE there is a number of different versions that can be implemented. To understand what can be used in hypothetical languages or implementations we will be categorizing them as:

- Constant expansion
- Macros
- Execution
- AST execution
- External execution

Constant expansion

An implementation defined as constant expansion is used primarily for numbers. While it does not form function execution during runtime, this is the basis for majority of the implementations.

All languages used this to some degree. Some have fancy support where by number constants can be in any number of formats. They could even form other types such as strings.

Design patterns

To fully take advantage of compile time functionality, it may not be possible to use design patterns that are well known. Normally they take advantage of well definated virtual interfaces to classes to enable swapping between implementations. This is a hinderence when working with compile time and meta-programming. For these situations you must know the exact implementation that you are deal with at all points. A great example of a common design pattern that can be adapted to compile time easily is the visitor pattern. The core difference is in the element instead of specifying the visitor abstraction interface, you use an abstract class as part of the meta programming arguments. An Example in the D programming language:

```
class ConcreteElement : Element {
	override void accept(T : Visitor)(T \
visitor) {
		visitor.visit(this);
	}
}
```

At this point it would be simple on Visitor to implement a specific implementation for the Element implementation. Alternatively a similar method to the one above could be used to handle it abstractly. Such as:

```
class ConcreteVisitor : Visitor {
	void visit(T : Element)(T obj) {
		pragma(msg, T.stringof);
	}
}
```

Of note is the runtime if statement support for determining if a class instance inherits from a specific interfaces/class.

```
if (Type to = cast(Type)from) {
        ...
}
```

This might be useful at compile time if the possible types were limited. However if the purpose is to generate code for runtime usage then checking type using a `static if` is the right tool.

Helper generators

This design pattern is a general use case one. The term generator is a rather awful designation, but it pushes the thought of what a generator does. It creates something to be used. Under this design pattern there is two types of functions. A helper function is a single purpose function that is designed to execute only at compile time. It mostly requires some form of compile time arguments via e.g. metaprogramming. It uses type information to derive new information and return it. Attributes and returning a specific piece of information from it are a great example. A generator function produces a function to be executed at runtime, but uses compile time information. This calls helper functions to do so.

References

- Cmsed, web service framework in D by the author:

  https://github.com/rikkimax/Cmsed
- Converge, CTFE support:

  http://convergepl.org/documentation/2.0/ctmp/
- Dvorm, ORM in D by the author:

  https://github.com/rikkimax/Dvorm
- Dylan, CTFE support:

  http://opendylan.org/books/drm/Macros
- Jade, templating language:

  http://jade-lang.com/
- The D programming language reference on traits expressions:

  http://dlang.org/traits.html
- The D programming language standard library reference for traits:

  http://dlang.org/phobos/std_traits.html

- Vibe.d, asynchronous IO framework: https://github.com/rejectedsoftware/vibe.d

Glossary

- API, Application Program Interface
- AST, Abstract Syntax Tree
- CEI, Compiler External Interface
- CTFE, Compile Time Function Execution (or Evaluation)
- JNI, Java Native Interface
- JVM, Java Virtual Machine
- ORM, Object Relational Model
- PHP, Hypertext Preprocessor
- UDA, User Defined Attribute
- UML, Unified Modeling Language