overload 131

FEBRUARY 2016 £3

Defining Concepts

Concepts provide a new way of constraining code. We see how to define and use them.

Maximising Discoverability of Virtual Methods

How C++11's `override` keyword can improve your code

Template Programming Compile Time Combinations & Sieves

A functional approach to generating sequences in C++

Classdesc: a Reflection System for C++11

An automated reflection system for C++

So Why Is Spock Such a Big Deal?

A history of testing on the JVM, and why Spock is so groovy

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 131

February 2016

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 132 should be submitted by 1st March 2016 and those for Overload 133 by 1st May 2016.

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of the ACCU
For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

Copyrights and Trade Marks

Be lucky

Do you consider yourself unlucky? Frances Buontempo wonders what we can do to avoid disasters.

Unfortunately, I have failed to write an editorial yet again. My hopeless, failed attempts may be auspicious though. We have previously considered failure being a potentially good thing. Perhaps we should now consider my misfortune. Could I be luckier next time? Am I just not trying hard enough? Does luck matter, or even mean anything? If people wish me "Good luck with that!" they may be stating I don't have a hope, or they may genuinely be 'wishing me luck'. Do wishes ever come true? Yes, they do, but you can't always tell in advance if you're onto a winner.

'Luck' seems to have its roots in a Dutch word 'gheluc' meaning fortune. To me this has a hint of ideas concerning gambling – "You win some, you lose some…" Some viewpoints might subscribe to an alternative stance taking on a notion of destiny or perhaps karma. "You reap what you sow." If you happen to be born into a fairly well-off family in a resource-rich country, you may have had the good fortune to be bought a PC at a young age, giving you the opportunity to learn to program in the privacy of your own bedroom. If you are the less fortunate other sibling, you might not get a chance to get anywhere near the aforementioned computer. On the other hand, if you don't have your own bedroom, don't have enough money for a computer, don't have an electricity supply, or perhaps have all these, but are told "You're not technical enough – you won't be any good at computers", you are jinxed, in a sense. Such a curse is the opposite of luck. Many a rational person would claim they don't believe in either. Yet the right circumstances can make a positive outcome more likely, while awkward circumstances make things harder. Some innovators are becoming aware of situations other than their own, so we see clockwork radios and computers, the Raspberry Pi, Code Clubs and so on. I have recently read several stories of women in Africa learning how to program which have left me amazed [TechWomen, Jjiguene, AWAP]. Even if fate hands you a lemon, you can make lemonade, to coin a phrase.

Different cultures have different ideas of luck. People will carry charms, which vary from place to place. I personally never understood how a rabbit's foot could be lucky. It certainly wasn't for the rabbit. These superstitions often have a long and interesting history, but sometimes their origins are lost in the mists of time. People will light joss sticks to appease the Gods. Or sacrifice some unlucky creature. I had also heard 'joss' used as a Chinese idea of luck, but was surprised to learn it comes from Portuguese 'deus', for God. As programmers, do we burn joss sticks, maybe just to mask code smells? Do we carry round a rabbit's foot, maybe an OS on a stick or similar, which could be a bit more use than a foot, even if it is the left hind foot of a bunny killed in a cemetery when there's a full moon? We probably all have rituals that we resort to, to make us feel 'luckier' or more hopeful of getting it right. Running tests, looking at the results of a static analyser, changing the whitespace and code layout to make it easier for us to read (while causing a massive fight with other team members), getting a greater variety of people to conduct code reviews in the vague hope of flushing out problems... The list is long. There is hope, and we constantly find new ways of making the situation more hopeful. If there's no hope, the people perish.

Can programmers ever be considered lucky or unlucky? Imagine for a moment you catch a multi-threading bug with some load testing in a development environment, thereby circumventing potential disaster for your customers, and your team. Is this lucky? In a sense, yes. In another sense, no, since running load testing in the first place was designed to flush out exactly this sort of issue. And yet, that you have heard of load testing is either lucky, or you have made an effort to read the right books or listen to the right talks or simply just have the imagination to find ways to make your life easier. Imagine instead, you don't have any automated tests round your code, try to manually test a few scenarios for a new feature and are told you must release this now because there's a deadline, and despite your protests, the release goes ahead. And then gets rolled back a few days later due to the bug reports. Is this unlucky? Perhaps. However, I suspect many readers would see this as a predictable and probably avoidable outcome, which happens surprisingly frequently. Why did 'the suits' not listen to your protests? Managing to make yourself understood is often about more than luck. Instead of complaining to people that an obscure edge-case may not work or that you can't possibly estimate how long it will take you to do something you've never done before in your life, it is better to try to understand what they are trying to achieve and what information they need. Linda Rising and Barbara Chauvin wrote an article a few years ago about the clash between programmers and managers in 'Using Numbers to Communicate – in the Spirit of Agile' [Numbers]. They suggested it helps to ease communication if you find a common way to talk, rather than perpetuating the 'us-and-them' attitude. In a fictitious scenario, they observe that if a programmer had "converted the issue to something numeric instead of personal, the issue would have been much clearer and easier for him [the manager] to understand the impact." Rather than complaining about meetings, keep a diary showing how many hours are spent in them. Then subtract how many hours have been saved by the face to face conversation. If it takes days to manually test a scenario comparing the numbers between today and two days ago, guestimate how long it might take to re-architect the code so you can artificially inject a different business date into the system. Then you can show if there will be a saving in programmer hours over the next budget period. Keep evidence, don't just rant. As you look at what you measure, you may even realise you were wrong and something completely different

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She works at Bloomberg, has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

is holding you up. This might make you look heroic, and then you may be listened to more in the future.

You can do a variety of things to increase your luck. Finding a good 'coding buddy' who you can turn to for help or advice designing a new system or troubleshooting an existing one is fortunate. Being a lone ranger who never gets a code review might make you look heroic, but is a lonely place to be. Why are no people working with you? Does your amazing talent frighten them off? Do you need to make an effort to be more approachable? If you are a newbie you can practise your craft. It is still worth practising whatever your skill level. A popular custom at the moment is coding katas, but just writing code might make you better. Getting the chance to pair program from time to time can be really useful. It might inspire you with new ways of working. It is often just a keyboard shortcut that speeds you up in the future. How auspicious. If you do feel isolated, you could join a geek group, for example the ACCU, just to have a new stream of ideas, thoughts and opinions. If you try to be nice to people, to listen, help and inspire you can grow your network in a useful way. People may start listening to you then. Sometimes it feels slower to work with other people, but it is often worth it. Just the knowledge sharing makes the process resilient in the face of staff turnover, illness and so on. Sometimes you might be wrong. Be willing to say "Oops" gracefully. Sometimes you might be right. Just stating your case tends not to bring on board any converts. If you can't convey your point with words, and stamping and finger pointing, then try a different approach. Drawing pictures can convey information, often more clearly than words or even tables of numbers. If you start to measure what's going on in your code-base, for example spotting files that change frequently but have no tests, this can be graphed easily. And no-one ever argues with facts, right?

You might find as you start measuring things your instinct was incorrect. Being honest is important. If you know you get stuck on something or are not very good at it, you can plug gaps in your knowledge. As mentioned, you could try code katas, read a book or write an article to get the idea clearer in your head. You could get a review from an expert in the subject. However, there is not enough time to become an expert at everything. It is ok to delegate to other people. If someone in your team is very good at something, allow them to get on with it. Do make sure they work with someone from time to time to avoid lone rangers and cowboys though. Perhaps they will give you an executive summary and you might learn a little yourself. You may find you have to do boring grunt work – fill in forms, grep log files, while the bright young things get on with the exciting stuff. That's OK. If you manage to do it without grumbling, or interfering with them too much, they may appreciate you. You might successfully automate the boring stuff while you are there, making everyone's lives better. Luck might therefore be about finding the right people to give you a leg up or a helping hand. "Stand on the shoulders of giants" as the saying goes. Or just frustration leads you to find a better way of working once you have figured out the root cause of the problem. The glass may be half empty, rather than half full. But that might be enough to motivate you to go to the bar and get a round in for everyone. Then your pint pot may overflow.

A person's environment or circumstances does affect the potential outcomes. However, this is not the full story. Difficult circumstances can drive innovation. Things being easy, even if not ideal, may stifle innovation. You may work in a business that is risk-adverse so suspect any great change is impossible, but even then you can make small things simpler. The saying about seeing a glass as half-empty or half-full I alluded to earlier is about attitude to the circumstances you find yourself in. It could be if you start to feel unlucky, then you sink into a downward spiral. You might start looking for things that are going wrong, and become paranoid. If this happens, you may find people notice, and they do start watching you. On the other hand, if you see a less than full glass as a chance to grab another pint, or make small but valuable tweaks to the process, code-base or team morale then the difference you have made might be noticed. Even if it's not noticed, which might be a good thing in an ultra-conservative environment, you have made your own life easier. Your attitude can influence your luck. Or your peace of mind. To quote Monty Python, "Let us not be down-hearted. One total catastrophe like this is just the beginning!"

Various studies have been conducted to analyse the perceived idea of luck. I am not convinced of the scientific status of any of these. Nonetheless, they can give pause for thought. For example, Richard Wiseman [Wiseman], writes about a study he conducted of 400 people who considered themselves to be either lucky or unlucky. He claims he found some distinct patterns in the two groups. Many of these revolved around attitudes. An important and believable point he made was lucky people followed their instincts, perhaps trusting themselves, while unlucky people expected trouble making themselves anxious in the process. If you don't believe in yourself, how can you expect anyone else to? Another point I got from the article regarded an experiment which asked how many pictures were in a magazine. The lucky people got it right, more quickly. There had been a statement part way through suggesting there were a given number of pictures in the magazine. The lucky people noticed and went with this, in the main. I can personally imagine not trusting this in the same way I might not trust a code comment that proudly claims "`//This is correct`". Yet, on other occasions, I will be on the look-out for clues. If I have a meeting coming up about some middleware I have never used before, I will catch a brief moment to read about it, so I can start thinking through what questions I need answered. I want to know the throughput **and** latency. Not just one. If I have an interview, on either side of the desk, and spot someone has an unusual name, I may search the internet to see if they have written an article or have some code up publicly. I can then have a more interesting exchange with them.

For one final thought, if everything does appear to be going wrong, and you consider yourself the unluckiest person in the world, I learnt one very valuable lesson in 2015. I attended the one-day *Agile in Banking* conference [AiB] where the closing keynote was given by Linda Rising. She was talking about *Fearless change* of course [Rising], which I must read one day. In the course of the talk she also told us about Maria's Rule – "there are very few problems that cake cannot solve." If everything appears to be a disaster, just bake the team a cake. Or buy donuts. This might be the best way to win friends and influence people. Then you may become the luckiest person in the world. The trick might be to spend time thinking about what you regard as lucky. As discussed, there are ways to increase your luck. What is the most unfortunate thing that's ever happened to you in your programming career? Perhaps you got sacked, but then ended up with a much better job. What's the luckiest thing that ever happened? Was it actually a close shave that you can avoid another time?

References

[AiB] Agile in Banking, http://agileinbanking.net/2015/

[AWAP] http://www.africanwomenadvocacyproject.org/PDFs/AWAP_Report_Jan%20201_Final.pdf

[Jjiguene] 'Women and girls are the answer to innovation in Africa' http://blogs.worldbank.org/nasikiliza/women-and-girls-are-answer-innovation-africa

[Numbers] 'Using Numbers to Communicate – in the Spirit of Agile' Rising and Chauvin, 2008, http://www.infoq.com/articles/rising-agile-spirit-numbers

[Techwomen] https://www.techwomen.org/feature-story/josephine-kamanthe-innovates-youth-education-with-solar-power-in-kenya

[Rising] *Fearless Change: Patterns for introducing new ideas* Addison Wesley 2004

[Wiseman] 'Be lucky – it's an easy skill to learn' June 2003 http://www.telegraph.co.uk/technology/3304496/Be-lucky-its-an-easy-skill-to-learn.html

Defining Concepts

Concepts provide a new way of constraining code. Andrew Sutton shows us how to define and use them.

This article is the second in a series that describe concepts and their use. In the first article, I describe how concepts are used to declare and constrain generic algorithms [Sutton15]. In this article, I discuss how to define and use concepts: the building blocks of the constraints used in the previous article. The next article will focus on systems of concepts, overloading, and specialization.

The features described in this article are based on the ISO Concepts Technical Specification (TS) [N4549], a formal extension of the C++ Programming Language. The specification is implemented in GCC and will be part of the forthcoming 6.0 release. Eric Niebler and Casey Carter are also working on a Ranges TS [N4560] that incorporates these language features and will define the base set of concepts needed for the C++ Standard Library.

Recap

In my previous article, I wrote about a simple generic algorithm, **in()**, which determines whether an element can be found in a range of iterators. Here is its declaration, modified slightly to suit the purposes of this article.

```
template<Range R, Equality_comparable T>
  requires Same<T, Value_type<R>>()
bool in(R const& range, T const& value);
```

The function **in** takes a **range** and a **value** as arguments. To specify the requirements on those arguments, the declaration uses three concepts:

- the type of the **range** must be a **Range**,
- the type of the **value** must be **Equality_comparable**, and
- the type of the **value** and that of the elements in the **range** must be the **Same**.

Value_type is not a concept. It is an alias of an internal type trait:

```
template<typename T>
using Value_type = typename value_type<T>::type;
```

We'll see how **value_type** can be defined later in this article.

Recall that the compiler internally transforms the concepts in the declaration into a single constraint. In order to use this function, any template arguments must satisfy this predicate:

```
Range<R>()
&& Equality_comparable<T>()
&& Same<T, Value_type<R>>()
```

```
error: cannot call function 'bool in(const R&, const T&)
[with R = std::vector<std::string>; T = char [6]]'
in(v, "Akron");
^
note: constraints not satisfied
in(R const& range, T const& value)
note: concept 'Same<char [6], std::string>()' was not satisfied
note: within the concept template<class T, class U> concept bool Same()
[with T = char [6]; U = std::string]
concept bool Same() { ... }
             ^~~~
note: 'char [6]' is not the same as 'std::string'
```

Figure 1

If this expression does not evaluate to **true** (given concrete template arguments for **R** and **T**), then the function cannot be called, and the compiler emits a useful error message. For example, compiling this program:

```
std::vector<std::string> cities { ... };
assert(in(cities, "Akron"));
```

will yield an error such as that shown in Figure 1.[1]

What exactly are **Same**, **Equality_comparable**, and **Range**, and how are they defined?

Concept definitions

A concept is a predicate on template arguments. In the Concepts TS, concepts are defined as a slightly simplified form of **constexpr** functions. Here is the declaration of **Same**:

```
template<typename T, typename U>
concept bool Same() { ... }
```

Concepts are defined by using the concept keyword in place of constexpr, and they must return bool. In order to make concepts simple to implement, fast to compile, yet sufficient to test properties of types, we impose a few restrictions on their definition:

- concepts must be defined at namespace scope,
- concepts cannot be forward declarations,
- concepts cannot take function arguments,
- concepts cannot be recursive,
- concepts cannot be explicitly specialized,
- concept definitions are limited to a single return statement, and
- the returned expression must be a logical proposition (i.e., convertible to bool).

1. This error is generated by GCC (compiled from trunk) with an extra patch (pending review) to improve concept checking diagnostics. The message has been modified for better presentation. Some type names have been shortened and the definition of **Same** is elided (...)

Andrew Sutton is an assistant professor at the University of Akron in Ohio where he teaches and researches programming software, programming languages, and computer networking. He is also project editor for the ISO Technical Specification, 'C++ Extensions for Concepts'. You can contact Andrew at asutton@uakron.edu.

The language syntactically limits concepts to simple logical propositions, but this isn't quite as restrictive as it sounds

The language syntactically limits concepts to simple logical propositions, but this isn't quite as restrictive as it sounds. Those propositions can evaluate any other constant expression. For example, here is the definition of the **Same** concept:

```
template<typename T, typename U>
concept bool Same() {
  return std::is_same<T, U>::value;
}
```

This concept expresses the requirement that two types must be the same. The concept is satisfied whenever **std::is_same<T, U>::value** is **true**. Of course, this concept is so fundamental and obvious that it may as well be defined by the compiler.

Concepts can also be defined as variable templates. For example, we could have defined **Same** like this:

```
template<typename T, typename U>
concept bool Same = std::is_same<T, U>::value;
```

Variable templates [N3615] were added to C++14 at the 2013 Bristol meeting, the same meeting at which the ISO Concepts TS was formally created. A variable template declares a family of variables whose values depend on template arguments. For example, the value of **Same** would depend on the types given for **T** and **U**.

Variable concepts are restricted in many of the same ways that function concepts are restricted:

- concepts must be defined at namespace scope,
- concepts cannot be explicitly or partially specialized, and
- the initializer expression must be a logical proposition.

Defining concepts in this way means that you can leave off the extra parentheses when using concepts in a **requires** clause:

```
template<Range R, Equality_comparable T>
  requires Same<T, Value_type<R>> // no parens!
bool in(R const& range, T const& value)
```

We've found that some developers prefer concepts to be declared and written this way despite the lack of overloading. The Concepts TS supports variable templates specifically because of this concern. Variable concepts were added to the TS only after variable templates were added for C++14. My preference is to define concepts as functions, so I use that style throughout this and the other articles in the series.

Syntactic requirements

While every type trait is potentially a concept, the most useful concepts are much more than simple wrappers. Think about **Equality_comparable**. It requires its template arguments to be usable with **==** and **!=** operators. In C++14, we might express those requirements using a conjunction of type traits or some other advanced mechanism. Listing 1 is a trait-based implementation. Here, **has_equal** and **has_not_equal** are type traits that rely on subtle use of language features to determine the availability of an expression for a type. Their definitions are not shown here.

```
template<typename T>
concept bool Equality_comparable()
{
  return has_equal<T>::value &&
    has_not_equal<T>::value;
}
```

Listing 1

This approach is both simple and powerful, yet indirect and totally inadequate to the task at hand. Using traits to state requirements obfuscates the intent, making concepts more difficult to read and write. It can also slow compilations, especially when the use of such constraints is ubiquitous throughout a library. More recent concept emulation techniques improve on readability [Niebler13], but we can do better still. The Concepts TS provides direct language support that makes writing concepts simpler, faster to compile, and allows the compiler to produce far better error messages.

To do this, we introduced a new kind of expression: the **requires** expression. Here is a complete definition of the **Equality_comparable** concept (see Listing 2). The **requires** keyword can be followed by a parameter list introducing names to be used to express requirements. Here, we have declarations of **a** and **b**.

The body of a **requires** expression is a sequence of *requirements*, each of which specifies one or more constraints for expressions and types related to a template argument. We refer to these as a concept's *syntactic requirements*.

In the **Equality_comparable** concept, both requirements are *compound requirements*, meaning they introduce multiple constraints: The expression enclosed within braces (e.g., **a == b**) denotes a constraint for a *valid expression*. When the concept is checked against a (concrete) template argument, the constraint is satisfied if the substitution of the template argument into the expression does not result in an error.

The trailing **-> bool** denotes an *implicit conversion constraint* on the result type of the instantiated expression. That constraint is satisfied only if the result is implicitly convertible to **bool**.

The **Range** concept has more interesting requirements. Let us define it in stages, starting with a first and naïve version (Listing 3). That is, a **Range** must supply a **begin()** and an **end()** function, each taking a **Range**

```
template<typename T>
concept bool Equality_comparable() {
  return requires (T a, T b) {
    { a == b } -> bool;
    { a != b } -> bool;
  };
}
```

Listing 2

A concept should include requirements for only the types and operations needed for its intended abstraction

```
template<typename R>
concept bool Range() {
  return requires (R range) {
    begin(range);
    end(range);
  };
}
```

Listing 3

argument. That's correct, but not every `begin()` and an `end()` function will do.

To be a `Range`, they must return input iterators:

```
requires (R range) {
  { begin(range) } -> Input_iterator;
  { end(range) } -> Input_iterator;
}
```

`Input_iterator` in another useful concept. When defining new concepts, we almost always build on a library of existing ones. `Input_iterator` is the representation in code of what is defined in English text in the ISO C++ standard.

When the type following the `->` is a concept name (or placeholder), the result type is deduced from the required expression. This is called an *argument deduction* constraint. If deduction fails, or if the deduced type does not satisfy the named concept, the constraint is not satisfied.

With this definition of `Range`, the result types of `begin()` and `end()` are deduced separately, which means that they can differ. This may not be your intent. As a general rule, if you have several operations that you intend to be the same type, give it a name:

```
requires (R range) {
  typename Iterator_type<R>;
    { begin(range) } -> Iterator_type<R>;
    { end(range) } -> Iterator_type<R>;
  requires Input_iterator<Iterator_type<R>>();
};
```

That is, `begin()` and `end()` must return the same type (here called `Iterator_type<R>`) and that type must be an `Input_iterator`. This last requirement is added by the nested `requires` clause within the body of the requires expression.

To be useful for our purposes, a `Range` must also name the type of its elements, its `Value_type`. For example, `in()` requires that the `Value_type` of its `range` is the same type as the type of its `value` argument. To complete the `Range` concept we require that it have a `Value_type` in addition to its `Iterator_type` (see Listing 4).

To ensure consistency, the value type of a range and its iterators must be the `Same`. Beyond that, however, there are no other requirements we want to make of `Value_type`. Those other requirements are imposed by algorithms. For example, the `in()` algorithm requires equality comparison, whereas `std::sort()` requires a total order. A concept

```
template<typename R>
concept bool Range() {
  return requires (R range) {
    typename Value_type<R>;     // Must have a
                                // value type.
    typename Iterator_type<R>; // Must have an
                                // iterator type.
    { begin(range) } -> Iterator_type<R>;
    { end(range) } -> Iterator_type<R>;
    // The iterator type must really be an
    // input iterator.
    requires Input_iterator<Iterator_type<R>>();
    // The value of R is the same as its
    // iterator's value type.
    requires Same<Value_type<R>,
              Value_type<Iterator_type<R>>>().
  };
}
```

Listing 4

should include requirements for only the types and operations needed for its intended abstraction. Including extra requirements can make a concept too strict (i.e., not broadly applicable).

When defining requirements for a concept, I introduce type requirements first, then simple and compound requirements, and nested requirements last. This is because constraint checking, the substitution of arguments into constraints to test for satisfaction, follows the short-circuiting logic of the `&&` and `||` operators. This means that failures detected earlier are less likely to result in unrecoverable instantiation failures later.

Ad hoc requirements

The use of alias templates to refer to associated types greatly reduces the verbosity of template declarations. Alias templates like `Value_type` and `Iterator_type` refer to facilities that compute associated types based on pattern matching on the 'shape' of the template argument. Listing 5 is a first naïve attempt to define `Value_type`.

This seems reasonable at first glance. However, we have not constrained the primary template of the trait definition, and that can cause problems. When the compiler selects the primary template for a template argument that does not have a nested `::value_type`, compilation will fail. This is an unrecoverable error that breaks concept checking.

We want to define the `value_type` trait so that it is instantiated if and only if there is a specialization that provides an appropriate type. To do this, we factor a new constrained specialization out of the primary template leaving it unconstrained and undefined (see Listing 6). Now, the `value_type` is defined only where it is meaningful. The new specialization is chosen only for classes that have a member called `value_type`.

Writing fundamental concepts requires an understanding of the way the type system and other language rules interact

```
template<typename T> struct value_type;

template<typename T>
using Value_type = typename value_type<T>::type;

// The value_type of a class is a member type.
template<typename T>
struct value_type {
  using type = typename T::value_type;
};

// The value_type of a pointer is the type of
// element pointed to.
template<typename T>
struct value_type<T*> {
  using type = T;
};

// The value_type of an array is its element type.
template<typename T, int N>
struct value_type<T[N]> {
 using type = T;
};
```

Listing 5

To avoid verbosity, I did not define a new concept like **Has_value_type**. Instead, I used a **requires** expression directly within the **requires** clause. Yes, **requires requires** is syntactically correct – it is not a typo. The first **requires** introduces the **requires** clause, the second starts the **requires** expression.

This syntax for ad hoc constraints is not optimized (i.e., gross) on purpose. Providing a more elegant syntax for these kinds of constraints might encourage programmers to think about generic code in terms of small syntactic fragments (although these are sometimes helpful when laying the foundations of higher level abstractions). In general, useful concepts have obvious and meaningful names.

Writing fundamental concepts requires an understanding of the way the type system and other language rules interact. For example, we cannot constrain the primary template directly because constraints are checked after name lookup. Every lookup for **T*** would fail because pointers do not have nested members. Libraries of concepts saves us from having to consider such subtleties all the time.

```
template<typename T>
struct value_type;

// The value_type of a class is a member type.
template<typename T>
  requires requires { typename T::value_type; }
struct iterator_type<T> {
  using type = typename T::value_type;
};
```

Listing 6

```
template<Range R, Equality_comparable T>
  requires Same<T, Value_type<R>>()
bool in(R const& range, T const& value) {
  for (Equality_comparable const& x : range) {
    if (x == value)
      return true;
  }
  return false;
}
```

Listing 7

When the type trait is instantiated during concept checking, the compiler considers each partial specialization, if none match (e.g., **int** is neither an array, nor does it have nested type names), then the compiler selects the primary template, which happens to be undefined. The result is a substitution failure that gets 'trapped' by the **requires** expression that causes the instantiation, and this causes enclosing concept to be unsatisfied.

In other words, **value_type** is a recipe for writing SFINAE-friendly type traits using concepts. The definition of the **Iterator_type** and its underlying trait have similar definitions.

Mixed-type requirements

Listing 7 is our working definition for the **in()** algorithm. As declared, the value type of **R** must be the same as **T**, which would make the following program ill-formed.

```
std::vector<std::string> cities { ... };
assert(in(cities, "Akron"));
```

A string literal does not have the same type as **std::string**, so the constraints are not satisfied. That's not good enough. The **std::string** class provides a number of overloads to make it work seamlessly with C-strings, and we should be able to use those in our generic algorithms. How can we change the algorithm to support these kinds of mixed-type operations?

We could redefine the algorithm so that **value** was a **Value_type<R>**. However, this would always require a conversion at the call site, which would almost certainly be a pessimization (converting a C-string to a **std::string** may require an allocation).

We could drop the **Same** requirement. But then the interface would not express how the elements in **range** are related to **value**, and we want our constraints to fully express the syntax used within the definition.

Our best choice is to change the **Same** requirement to something more permissive: a concept that supports equality comparisons between values of different types. Rather creating a concept with a different, name we can extend **Equality_comparable** by adding a new definition that takes two arguments instead of one. That is, we overload the

The ability to extend a concept to support mixed-type requirements is an essential tool for making algorithms more broadly applicable

```
template<typename T, typename U>
concept bool Equality_comparable() {
  return requires(T t, U u) {
    { t == u } -> bool;
    { u == t } -> bool;
    { t != u } -> bool;
    { u != t } -> bool;
  };
}
```

Listing 8

`Equality_comparable()` function. That concept must express requirements for all the ways in which we can compare values of different types for equality (see Listing 8).

This concept requires the symmetric comparison of values of type `T` and `U`.

We can now use the mixed-type `Equality_comparable` concept to weaken the constraints on the `in()`.

```
template<Range R, Equality_comparable T>
  requires Equality_comparable<T, Value_type<R>>()
bool in(R const& range, T const& value);
```

These constraints fully specify the syntax used within the implementation, the program compiles as expected, and it does not introduce any additional runtime overhead. This is a better declaration of `in()`; it's also the version we used in the first article. The ability to extend a concept to support mixed-type requirements is an essential tool for making algorithms more broadly applicable, without extra notational or runtime overheads. The Palo Alto report, for example, uses this technique for total ordered types, all binary relations, and all binary operations.

These extended definitions are not available for variable concepts because the capability is based on function overloading. This is not a limitation imposed by concepts; you simply cannot overload variables in C++.

Semantic requirements

The syntactic requirements of a concept only tells us what expressions and associated types can be used with a template argument (or template arguments). In general, we would very much like to know what those expressions and types actually mean. Just as importantly, it would be helpful for the compiler and other tools to be able to reason about the meaning of such expressions in order to support optimization and verification. Unfortunately, the Concepts TS does not provide direct language support for writing semantic requirements. Instead, we must rely on conventional forms of documentation to specify the semantics of operations operations.

C++0x concepts supported a feature called 'axioms', but it was added late in the development of C++11 [N2887], and their utility had not been fully explored by the time concepts were removed. Axioms were also major feature of the Palo Alto report [N3351]. However, as the proposal for Concepts Lite evolved, the Concepts Study Group (SG8) decided to leave axioms out pending further exploration. There is ongoing research related to compile-time checking of semantic requirements [DosReis09], so we hope to see axioms in the future.

Conclusions

Concepts are fundamental building blocks for our thinking and for our code; they provide the foundation upon which we design and implement software. The Concepts TS provides direct language support for the specification of concepts and their syntactic requirements. However, we must not forget or downplay the importance of the semantic aspects of concepts. A concept without semantics is merely a snippet of code.

In the next article, I will discuss systems of concepts, and how overloading and specialization based on constraints can be used to select optimal algorithms at compile time. ■

Acknowledgements

The design of the features in the Concepts TS was the result of collaboration with Bjarne Stroustrup and Gabriel Dos Reis. That material is based upon work supported by the National Science Foundation under Grant No. ACI-1148461. Bjarne Stroustrup also provided valuable feedback on drafts of this paper.

The WG21 Core Working group spent many, many hours over several meetings and teleconferences reviewing the Concepts TS design and wording. This work would not have been possible without their patience and attention to detail. Many people have submitted pull requests to the TS or emailed me separately to describe issues or suggest solutions. I am grateful for their contributions.

I would also like to acknowledge all of the early adopters of the GCC concepts implementation. Their feedback (often in the form of bug reports) has been invaluable.

References

[DosReis09] Dos Reis, G. 'A System for Axiomatic Programming' *Lecture Notes in Compute Science*. Vol. 7362. 2012. pp 295-309.

[N2887] Dos Reis, G., Stroustrup. B., Merideth, A. 'Axioms: Semantics Aspects of C++ Concepts' ISO/IEC WG21 N2887, Jun 2009.

[N3351] Stroustrup, B., Sutton, A. (eds). 'A Concept Design for the STL' ISO/IEC WG21 N3351, Feb 2012.

[N3615] Dos Reis, G.. 'Constexpr Variable Templates' ISO/IEC WG21 N3615, Mar 2013.

[N4549] Sutton, A. (ed). ISO/IEC Technical Specification 19217. 'Programming Languages – C++ Extensions for Concepts', Aug 2015.

[N4560] Niebler, Eric, Carter, C. Working Draft, 'C++ Extensions for Concepts', ISO/IEC WG21 N450. Nov 2015. pp. 213.

[Niebler13] Niebler, E. 'Concept Checking in C++11' 23 Nov 2013. Web.

[Sutton15] Sutton, A. 'Introducing Concepts' ACCU *Overload*. Vol 129. Oct 2015. pp. 4–8.

On Zero-Side-Effect Interactive Programming, Actors, and FSMs

Functional programming is alien to many programmers. Sergey Ignatchenko considers parallels between actors and finite state machines.

Greetings Earthlings! We come in peace. Take me to your leader...
~ Aliens

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

NB: Please don't expect to find any Big Computer Science Truths within this article; all the things mentioned here are either well-known or should be well-known to computer science people. It is the question 'how to make functional programming more usable in industry' which this article is about. Also note that I'm coming from the imperative programming side, so don't hit me too hard if I use terminology which is unusual in functional programming circles.

Functional programming introduces quite a few interesting concepts (personally, I am a big fan of pure functions). However, whether we like it or not, functional programming languages (especially 'pure' ones such as Haskell) are very much out of mainstream use, at least within that part of the industry which deals with *interactive programming* (more on *interactive programming* below).

This utter lack of popularity outside of *computational programming* is a trivial observation, hardly worth any further discussion, except for the question "WHY are functional programming languages not popular?" Of course, any explanation which says that there is something wrong with a language inevitably invites brutal lashing by its hardcore zealots (who will say that the problem is not with the language, and that it has all the necessary features,[1] it is just idiots like me who don't understand the beauty of it all). On the other hand, to start improving things one needs to realize what the problems are, so I will take the risk of being pounded in the name of the 'greater good' (the one of pure functions getting into industry programming).

On computational programming and interactive programming'

IMHO, one of the big reasons behind the lack of popularity of functional languages is that for functional programming,[2] there is a significant difference between two well-known entities, which I will call *computational programming* and *interactive programming*.

Computational programming

Let's define *computational programming* as the process of traditional calculations, with lots of input data coming in, and some results coming out. For *computational programming*, the result is completely defined by inputs (with inputs not allowed to change while we're computing), and all we need is to calculate an arbitrarily complex function

```
F(INPUTS[])
```

where `F` is our function, and `INPUTS[]` is a vector of inputs.

This is the area where functional programming really shines. All the functions can be made 'pure functions' in a very natural manner ('natural manner' being a synonym for 'a manner easily understandable by subject matter experts'), with no real need to assign variables, and all the other resulting goodies. This is to be expected, as it is also very natural for academics (who're traditionally positively in love with formulas), so no wonder that they've designed something optimized for their own needs. BTW, I don't mean that such an optimization is a bad thing *per se*. However, when we're using something optimized for one field, using it in another field requires us to conduct applicability analysis.

From practical perspective, *computational programming* applies to such fields as HPC.

HPC

High Performance Computing (HPC) most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business. [HPC]

Interactive programming

Most of out-of-academia (and out-of-HPC) programming cannot easily be described by merely taking pre-defined inputs and producing outputs. Let's take a very simple (but immensely practical) example. Let's consider a system which has two users, and needs to give something (let's say, a cookie) to the user who comes first. This task, while being extremely simple, illustrates the huge difference between *computational programming* and *interactive programming*. In particular, we cannot possibly decide which of the users will get the cookie until after we've launched our program.

Describing this kind of logic in terms of imperative programming is trivial, while describing it in terms of functional programming is much less obvious. Strictly speaking, to solve an interactive problem such as above via functions, we'd need to write a function

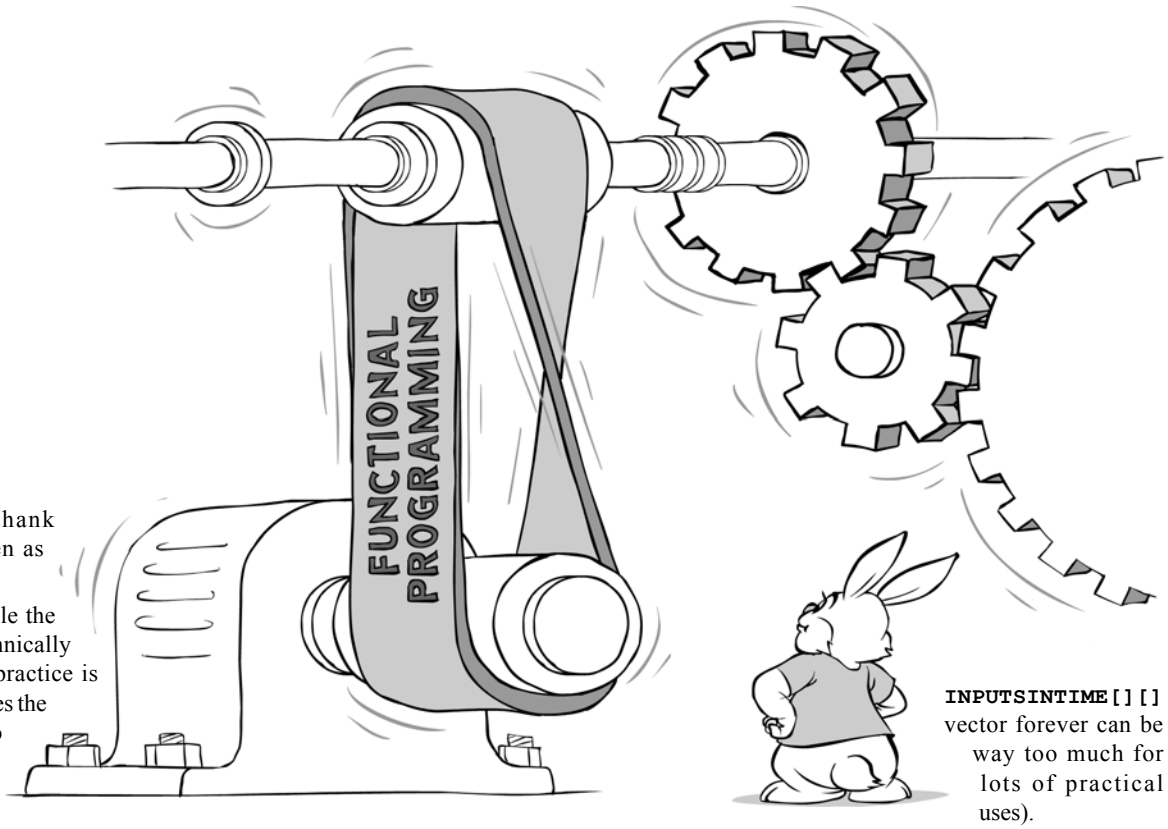
```
F'(INPUTSINTIME[][])
```

with vector `INPUTSINTIME` of inputs being two-dimensional, with the first dimension being the same as that for `INPUTS[]`, and the second

1. Of course, they are. All Turing-complete programming languages do have all the necessary features, and this includes the Brainfuck programming language. The only difference between the languages is how convenient is it to use the language.
2. Actually, the difference stands for any kind of programming paradigm, but imperative languages tend to hide the differences with more skill than functional ones.

'No Bugs' Hare Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com



dimension being time (thank goodness, time can be seen as discrete for our purposes).

Now, let's observe that while the `F'()` representation is technically correct, dealing with it in practice is really awkward. Not only does the logic itself have nothing to do with the terms in which the task is defined, but we're also required to keep all the history of all the `INPUTSINTIME` vector to run our program, real ouch!

One common solution to deal with this problem practically is to introduce (in addition to pure functions) some 'actions'. However, I don't want to accept 'actions' as a viable solution for the business-logic parts of the code, as they re-introduce side effects, defeat all the purity-related goodies, and essentially bring us back to the side-effect-ridden world :-(.

Alternatively, for the vast majority of cases (if not for all cases) we can rewrite our function `F'()` into a much easier to deal with form, by defining a finite state machine (more precisely – an ad hoc finite state machine, as defined in [Calderone13] and [NoBugs15a]) with a transition function `T(S,INPUTS[])` (with `T()` taking current state `S` and current `INPUTS[]` and returning new state `S'` back). Then, we can rewrite `F'(INPUTSINTIME[][])` as follows:

```
F'(INPUTSINTIME[][]) =
  T(...T(T(S0,INPUTSINTIME[][0]),
                INPUTSINTIME[][1])...)
```

where `S0` is an initial state of our state machine.

It means that to design an interactive program in a really pure function style, we don't need to define the really awkward `F'(INPUTSINTIME[][])`,[3] but can define the easily understandable transition function `T(S,INPUTS[])` (which can and should be a 'pure function'). Now, compared to our `F'()` function, we have come much closer to having a 'natural' representation of our interactive program.[4] In fact, function `T()` is very similar to the way interactive (and especially distributed) programs are usually written in a imperative programming language: as an event-driven program (which is equivalent to an ad hoc state machine).

BTW, implementation-wise, when a programming language knows that our function `T` is going to be used as a part of state machine, it can easily discard all the states except for current one from the cache, as well as discard all the previous values of `INPUTSINTIME[][]` provided that it always carries the last state. On the other hand, technically, it is no more than optimization compared to our purely functional completely side-effect-free function `F'()`, albeit a very important one (as storing all the `INPUTSINTIME[][]` vector forever can be way too much for lots of practical uses).

On the other hand, let's mention that while it is technically possible to describe our interactive system in terms of pure functions (see `F'()` above), it is strictly necessary to store some kind of historical state for our system; it may take the form of storing prior inputs (`INPUTSINTIME[]`), or may take the form of storing `S`, but for any interactive system we cannot possibly live without some kind of state which changes over time. And given the choice between storing `INPUTSINTIME[]` and storing `S`, I certainly prefer storing `S`.

FSM-based actors: a very practical observation

In practice, even before I learned about functional programming, I found myself using (and forcing my teams to use as part of system-wide architecture) 'actors' based on ad hoc finite state machines for large-scale distributed systems; here I mean 'actor' as a thread, a queue, and a deterministic Finite State Machine (FSM, a.k.a. finite-state automaton).[5] I've found that this Actor/FSM approach has lots (and I mean LOTS) of very practical benefits (described, in particular, in [NoBugs15a]). Just to give one extremely practical example described in [NoBugs15a], such deterministic FSMs allow the holy grail of production post-mortem to be achieved(!).

The programming style of these ad hoc finite state machines, while imperative, has almost universally used the pattern in Listing 1.

Moreover, I've found myself arguing to make the `process_event()` function entirely deterministic (by moving all the time-related and other system-related stuff out of the function, see [NoBugs15a] for further discussion), which brings the `process_event()` very close to being a 'pure' function (in a sense that it doesn't have side effects, with side effects defined quite loosely but still guaranteeing 100% determinism).

Now, if we take a closer look at this pattern, we'll see that it can be converted into a functional-style function `T()` in a trivial manner, simply by making a transition function `T` out of the `CHECK INPUTS` and `CALCULATE CHANGES` parts of `process_event()` (and the `APPLY CHANGES` part will stay outside our function `T()` and will be made by the compiler or something).

3. Nor to define 'actions' with side effects.
4. When compared to 'actions', we are still completely within the 'pure function' world(!)
5. I don't want (nor I am really qualified) to go into discussions whether this interpretation of Actors is a strict Actor concurrency model as defined by Hewitt [Hewitt73]; it does, however, correspond to 'actors' as implemented by Erlang and Akka, which I think is more important for our purposes.

```
class FSM {
  State s;
  void process_event(Event& event) {
   // CHECK INPUTS
   //   check validity of event,
   //   taking into account s
   //   DO NOT modify s
   //   may throw an exception
   //   semantics of thrown exception is obvious
   //     as s is not modified
   ...

   // CALCULATE CHANGES
   //   calculate changes which are to be made
   //   DO NOT modify s (yet)
   //   may throw an exception
   //   semantics of thrown exception is still
   //     obvious
   ...

   // APPLY CHANGES AND SEND OUTGOING EVENTS
   //   modify s
   //   no exceptions thrown, as semantics can
   //     become difficult to manage
  }
};
```

Listing 1

Let's take our very simple example of a stateful program, which needs to give a cookie on a first-come first-served basis.

Then, in usual C++ code, this might look like Listing 2.

The separation between different parts of the **process_event()** is not 100% clear, but this can be easily rewritten into Listing 3.

Note that here all substantial logic resides within the side-effect-free function **T()** (which is actually nothing but a classic FSM transition function), and that **process_event()** can actually be written in a very generic manner. More importantly, this second C++ form is very close to pure functional programming. In fact, in Haskell, the very same transition function **T()** can be defined along the lines shown in Listing 4, which is probably not the best Haskell but is hopefully sufficient to tell the story.

While syntactically Haskell's **t()** looks quite different from **C++::T()**, semantically they're strictly equivalent. For example, for the **C++::T()** **give_cookie** variable, the Haskell version has an equivalent function

```
class FSM {
  bool still_have_cookie;
  FSM() {
    still_have_cookie = true;
  }

  void process_event(const Event& ev) {
   // we assume that the only event which can come,
    // is request for a cookie
    // CALCULATE CHANGES
    bool give_cookie = still_have_cookie;

    // APPLY CHANGES
    if(give_cookie) {
      post_event(GIVE_COOKIE_ID);
      still_have_cookie = false;
    }
    else {
      post_event(NO_COOKIE_ID);
    }
  }
};
```

Listing 2

```
void process_event(const Event& ev) {
  pair<int,bool> event_and_new_state = T(ev);
  //APPLY CHANGES
  still_have_cookie =  event_and_new_state.second;
  post_event(event_and_new_state.first);
}

pair<int,bool> T(const Event& ev) const {
  //we still assume that the only event which
  //can come, is request for a cookie
  //CALCULATE CHANGES
  bool give_cookie = still_have_cookie;

  //STILL CALCULATE CHANGES
  int event_to_be_posted;
  new_still_have_cookie = still_have_cookie;
  if(give_cookie) {
    event_to_be_posted = GIVE_COOKIE_ID;
    new_still_have_cookie = false;
  }
  else {
    event_to_be_posted = NO_COOKIE_ID;
  }
  return pair<int,bool>(event_to_be_posted,
                        new_still_have_cookie);
}
```

Listing 3

give_cookie (which, when called, will return exactly the same value which **C++::give_cookie** variable will have when the C++ code is executed). The same stands for each and every variable from **C++::T()**, all the way to the returned **C++ pair<>** (and the Haskell tuple).

And note that the Haskell version is functional in its purest form – as there is no I/O, there is no need for actions, no sequencing, no monads, etc.

This similarity in semantics between imperative event-driven programming and functional programming (I hope) means that in fact, industry developers (especially those who need to deal with distributed systems) are already more or less prepared to write state machines in a style that is very close to functional (a 'pure function' style), the only remaining (though admittedly large) challenge being how to wrap this style into more familiar syntax.

Another way to put this observation (which is the key point I want to make in this article) is the following.

```
t :: State -> Event -> ([Event],State)

give_cookie :: State -> Bool
give_cookie st = still_have_cookie st
event_to_be_posted st ev
  = if give_cookie st then [Event(give_cookie_id)]
else [Event(no_cookie_id)]
new_still_have_cookie st ev
  = if give_cookie st then State(False) else st

t st ev = ( event_to_be_posted st ev,
            new_still_have_cookie st ev )

-- t() just calculates and returns (new_state,
-- list_of_events_to_be_posted)
-- process_event() (not shown) will need to have
-- side effects, but can be written in a
-- completely generic manner and kept completely
-- out of sight so our FSM code is completely
-- side-effect free
```

Listing 4

Ad hoc deterministic Finite State Machines is an *interactive programming* style, which can be understood (and reasonably expressed) by both industry-oriented imperative programmers, and functional programmers.

On Actors, concurrency, and interactive programming

Usually, Actors are considered as merely a way to achieve concurrency. I am arguing that they have a much more important role than just doing that: I see Actors (based on deterministic FSMs) as *the* way to implement *interactive programs* as defined above. And as *interactive programming* is what the vast majority of the industry developers are doing out there, making writing such Actors convenient is important for the industry to start benefiting from some of functional programming concepts (specifically – from 'pure functions' and determinism).

On Actors and composability

In [Chiusano10], it is argued that actors are not 'composable',[6] which impedes their use for concurrency purposes. If this is the case, it would mean that indeed actors are not so good for practical use. Let's take a closer look at this problem.

Actually, the author admits that actors can be made composable, but he argues that in this case they will be just an inferior form of pure functions. While this observation does stand for *computational programming* as described above (and where all the examples provided by the author belong), it doesn't stand for *interactive programming*. With *interactive programming*, even if we have our Actor as completely deterministic, it still inherently has some kind of state (see above), so it won't degenerate into a mere 'pure function'.

In other words, I tend to agree with [Chiusano10] that using Actors to describe concurrency for *computational programming* might be not that good an idea and that pure functions describe what we need for concurrency better (in theory, that is, while we can leave practical considerations such as compilers able to compile these things properly to others).

However, using Actors for *interactive programming* is a very different story, and that's where (stateful!) Actors really shine. It is confirmed by real-world experiences, starting from the practical use of Erlang for building large systems (with billions of transactions per day), continuing with my own experience with C++ FSMs for a highly interactive system (processing around a billion messages per day), and the recent wave of popularity for Akka Actors. IMNSHO, deterministic Actors are the very best thing in existence for *interactive programming*, with lots of very practical benefits (from replay-based regression testing and production post-mortem described in [NoBugs15a], to protection of in-memory state against server faults as described in [NoBugs15b], with no need to deal with thread sync and business logic at the same time, in between).

Bottom line

With this article, I actually want to emphasize two points. The first one is more of theoretical nature (and it is known but not often articulated). It is that deterministic FSMs (or 'actors' with the event processing function being 'pure') can be seen as a flavour of functional programming with a quite specific caching.

The second point is of more practical nature. There are certainly some very useful things which are already carried from functional programming into industry; in particular, its pure functions and determinism. However, to simplify adoption of functional languages for *interactive programming* which forms a huuuge part of the industry, I (as a guy coming from industry), would like to see the following:

- explicit and extensive discussion about the differences between *computational programming* and *interactive programming*. These two are very different beasts, and mixing them together causes lots of confusion. While the difference can be found in literature, it is by far not as prominent as it should be
- support for ad hoc FSMs (a.k.a. event-driven programs) as 'first-class citizens' (of course, with transition function being 'pure'). While it is possible to program a finite-state automaton in most (all?) functional programming languages, the syntax is usually ugly and confusing (for example, creating a new instance of FSM on each iteration certainly doesn't look 'natural' enough for subject matter experts; and 'actions' seem to go against the very nature of functional programming; an explicit 'transition function' will do much better in this regard). Event-driven programs are very common for interactive stuff in the industry, and can be explained very easily to the guys coming from there. On the other hand, given that *interactive programming* covers the vast majority of industry programs, explicitly supporting it (and spending time on making it very easy to use) makes perfect sense (of course, this only makes sense if there is a desire to proliferate functional programming beyond academia and HPC). ■

References

[Calderone13] Jean-Paul Calderone, 'What is a State Machine?'

[Chiusano10] Paul Chiusano, 'Actors are not a good concurrency model', http://pchiusano.blogspot.com/2010/01/actors-are-not-good-concurrency-model.html

[Hewitt73] Carl Hewitt; Peter Bishop; Richard Steiger (1973). 'A Universal Modular Actor Formalism for Artificial Intelligence'. IJCAI.

[HPC] http://insidehpc.com/hpc-basic-training/what-is-hpc/

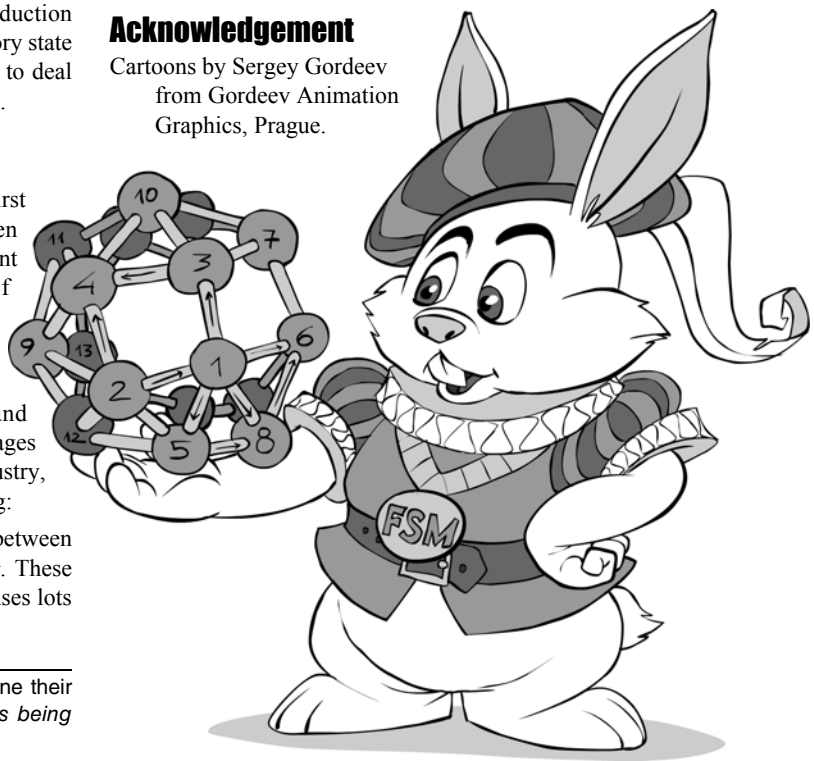
[Loganberry04] David 'Loganberry' Buttery, 'Frithaes! – an Introduction to Colloquial Lapine', http://bitsnbobstones.watershipdown.org/lapine/overview.html

[NoBugs15a] 'No Bugs' Hare, 'MMO Modular Architecture: Client-Side. On Debugging Distributed Systems, Deterministic Logic, and Finite State Machines'. http://ithare.com/chapter-vc-modular-architecture-client-side-on-debugging-distributed-systems-deterministic-logic-and-finite-state-machines

[NoBugs15b] 'No Bugs' Hare, 'Server-Side MMO Architecture. Naïve, Web-Based, and Classical Deployment Architectures', http://ithare.com/chapter-via-server-side-mmo-architecture-naive-and-classical-deployment-architectures/

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.



6. Entities are composable if we can easily and generally combine their behaviours in some way *without having to modify the entities being combined*.

Template Programming Compile Time Combinations & Sieves

Functional style frequently uses sequences. Nick Weatherhead applies these ideas to combinations in C++.

Previously [Weatherhead15], I discussed the use of C++ templates to compile time program some well-known string katas. Template metaprogramming in this way imposes a functional style with the immutability of variables a key consideration. In this article I'm applying the same treatment to combinations of k elements and a variation on Eratosthenes sieve. The foundations for each are built as we go along including holding data within lists, adding and removing items from them, implementing queues, defining sequences and operating on them with sets. There is a bit to it so you may wish to skip ahead to a solution and come back to the detail. Either is fine but you'll probably want to keep Listing 1 to hand for reference.

Lists

Without arrays and mutability at our disposal, a structure is required that can represent a list and after each operation produces a new variant. It's common for functional languages to have an operation which adds an element to the beginning of a list and is typically known as cons (short for construct). This functional list differs from the imperative viewpoint of linearly linking elements with a head at one end, a body and a tail at the other. Instead it is woven from a compound pair of pairs. Each pair comprises a head plus tail that splits until a null tail terminates a given path. It's good to have an appreciation of its recursive nature and the other features it affords see [SICP96]. However, for the most part, this article will attempt to adhere to the metaphor for a single chain.

In Listing 1, `list_` (42–179) takes a `HEAD` element and `TAIL` elements; an additional parameter `DELIM` defines the separator used between elements when printing. Appending an underscore to a class's name is the convention used here to indicate internal usage, hence, `list` (181–183) exposes `list_`. No-operation `nop` (5–11) simply places a null marker into the output as a guard element. A specialisation of `list_` (185–191) which has `nop` as its head element is used to indicate that it's a list of lists and when printed each list can be delimited by another character. Another `list` (193–207) is provided for elements at the end of a list i.e. that's `TAIL` is `nil` (13–40) where some functions are overridden whilst `nil` implements functions that operate on an empty list. When performing operations between elements it is convenient to think in terms of what appears on the right and left hand sides, so `list_` provides some internal functions that take `RHS` and `LHS` respectively. A value of type `v` (209–212) is itself a `list_` with its `HEAD` being a single value element of the same type. Thus each element can be operated on in the same way.

Queues

Imperative lists and vectors describe consecutive elements, as does the functional list, and all can be used as the underlying structure of a queue. Removing the `HEAD` of a `list_` requires a simple reference to the `TAIL` elements. To `prepend` (28–29, 61–62, 205–206) elements another `list_` is created with the new content in the `HEAD` and the existing content in the `TAIL`; `append` (28–29, 59–60, 203–204) just reverses the concatenation of the `HEAD` and `TAIL`. There are also implementations of these that take

```
 1 #include <iostream>
 2
 3 using namespace std;
 4
 5 struct nop {
 6
 7   static const char delim = '\0';
 8
 9   friend ostream& operator<<( ostream& os
10   , const nop& ) { return os; }
11 };
12
13 struct nil : nop {
14
15   typedef nil type;
16
17   template< size_t SIZE = 0 > struct size {
18
19     static const size_t value = SIZE;
20
21     friend ostream& operator<<( ostream& os
22     , const size& ) { return os << SIZE; }
23   };
24
25   template< class RHS >
26   struct append  : RHS { typedef RHS type; };
27   template< class RHS >
28   struct prepend : append< RHS > { };
29   template< class RHS >
30   struct uunion  : append< RHS > { };
31   template< class RHS >
32   struct except  : nop { typedef nil type; };
33   template< class RHS >
34   struct intrsct : except< RHS > { };
35
36   template< size_t N, size_t I_SIZE, class O
37   , size_t O_SIZE = 0 > struct kombine_
38   : conditional< O_SIZE == N, O, nop >::type
39   { };
40 };
41
```

Listing 1

Nick Weatherhead Nick's first encounter with programming was copying lines of code from magazines into the now venerable family BBC B. His teacher persuaded him to take computer science during his first term of A-Levels. This led to many hours of puzzle solving and programming, a relevant degree and finally gainful employment within London's financial sector. You can contact Nick at weatherhead.nick@gmail.com

When performing operations between elements it is convenient to think in terms of what appears on the right and left hand sides

```
42 template< class HEAD, class TAIL
43 , char DELIM = ',' > struct list_ {
44
45   #define t typename
46
47   typedef list_  type; typedef list_  LHS;
48   typedef HEAD   head; typedef TAIL   tail;
49
50   static const char delim = DELIM;
51
52   friend ostream& operator<<( ostream& os
53   , const list_& ) { return os
54     << HEAD( ) << tail::delim << TAIL( ); }
55
56   template< size_t SIZE = 0 > struct size
57   : tail::template size< SIZE + 1 > { };
58
59   template< class RHS > struct append
60   : list_< LHS, RHS, DELIM > { };
61   template< class RHS > struct prepend
62   : list_< RHS, LHS, DELIM > { };
63
64   template< class LHS, class RHS >
65   struct append_
66   : LHS::type::template append< RHS >  { };
67   template< class LHS, class RHS >
68   struct prepend_
69   : LHS::type::template prepend< RHS > { };
70   template< class LHS, class RHS >
71   struct except_
72   : LHS::type::template except< RHS >  { };
73   template< class LHS, class RHS >
74   struct uunion_
75   : LHS::type::template uunion< RHS >  { };
76   template< class LHS, class RHS >
77   struct intrsct_
78   : LHS::type::template intrsct< RHS > { };
79
80   template< class RHS, bool = true >
81   struct uunion
82   : conditional< ( LHS::head::value <
83                    RHS::head::value )
84     , append_<   LHS::head
85       , uunion_< LHS::tail,   RHS       > >
86     , append_< t RHS::head
87       , uunion_< LHS      , t RHS::tail > >
88     >::type {
89       typedef HEAD head; typedef TAIL tail; };
90
91   template< bool NA > struct uunion< nil, NA >
92   : append_< head, tail > { };
```

Listing 1 (cont'd)

```
93
94   template< class RHS, bool = true >
95   struct intrsct
96   : conditional< is_same< LHS, RHS >::value
97     , LHS
98     , t conditional< ( LHS::head::value ==
99                        RHS::head::value   )
100      , append_< LHS::head
101        , intrsct_< LHS::tail, t RHS::tail > >
102      , t conditional< ( LHS::head::value <
103                         RHS::head::value  )
104        , intrsct_< LHS::tail,   RHS     >
105        , intrsct_< LHS      , t RHS::tail >
106        >::type
107      >::type
108    >::type {
109      typedef HEAD head; typedef TAIL tail; };
110
111  template< bool NA >
112  struct intrsct< nil, NA > : nil { };
113
114  template< class RHS, bool = true >
115  struct except
116  : conditional< is_same< LHS, RHS >::value
117    , nil
118    ,t conditional< ( LHS::head::value <
119                      RHS::head::value  )
120      , append_< t LHS::head
121        , except_< LHS::tail,   RHS       > >
122      , t conditional< ( LHS::head::value >
123                         RHS::head::value )
124        , except_< LHS      , t RHS::tail >
125        , except_< LHS::tail, t RHS::tail >
126        >::type
127      >::type
128    >::type {
129      typedef HEAD head; typedef TAIL tail; };
130
131  template< bool NA > struct except< nil, NA >
132  : append_< head, tail > { };
133
134  template< size_t N, size_t I_SIZE
135  , class O, size_t O_SIZE > class kkombine_ {
136
137    friend ostream& operator<<( ostream& os
138    , const kkombine_& that ) {
139
140      static const int i_size = I_SIZE - 1;
141
142      return os
143      << t TAIL::template kombine_< N, i_size
```

Listing 1 (cont'd)

```
144         , prepend_< HEAD, O >, O_SIZE + 1 >( )
145       << t TAIL::template kombine_< N, i_size
146         , O                  , O_SIZE     >( );
147   } };
148
149   template< size_t N, size_t I_SIZE, class O
150     = nop, size_t O_SIZE = 0 > struct kombine_
151   : conditional< !O_SIZE && I_SIZE == N
152     , prepend_< type, nop >
153     , t conditional< O_SIZE == N, O
154       , t conditional< !I_SIZE, nil
155         , kkombine_< N, I_SIZE, O, O_SIZE >
156         >::type
157       >::type
158     >::type { };
159
160   template< size_t N > struct kombine
161   : kombine_< N, size< >::value > { };
162
163   template< size_t N, bool NA = true >
164   struct powerset_ {
165
166     friend ostream& operator<<( ostream& os
167     , const powerset_& that ) { return os
168       << powerset_< N - 1 >( )
169       << kombine< N >( ); }
170   };
171
172   template< bool NA >
173   struct powerset_< 0,  NA > : nop { };
174
175   struct powerset
176   : powerset_< size< >::value > { };
177
178   #undef t
179 };
180
181 template< class HEAD, class TAIL
182 , char DELIM = ',' > struct list
183 : list_< HEAD, TAIL, DELIM > { };
184
185 template< class TAIL, char DELIM >
186 struct list_< nop, TAIL, DELIM > : TAIL {
187
188   friend ostream& operator<<(
189     ostream& os, const list_& ) {
190     return os << ' ' << TAIL( ); }
191 };
192
193 template< class HEAD, char DELIM >
194 struct list< HEAD, nil, DELIM >
195 : list_< HEAD, nil, DELIM > {
196
197   typedef HEAD type;
198
199   friend ostream& operator<<(
200     ostream& os, const list& ) {
201     return os << HEAD::value; }
202
203   template< class RHS > struct append
204   : list< HEAD, RHS, DELIM > { };
205   template< class RHS > struct prepend
206   : list< RHS, HEAD, DELIM > { };
207 };
208
209 template< class T, T V, class VS, char DELIM >
210 struct v
211 : list< v< T, V, nil, DELIM >, VS, DELIM > {
212   static const T value = V; };
```

Listing 1 (cont'd)

explicit left and right hand side arguments – see **append_** (65–66) and **prepend_** (67–78).

Size

Functional programs don't have loop constructs so rely on recursion to perform inductive operations. Templates don't optimise tail recursive calls so each grows the stack, hence compilers place a limit on its depth (note that despite this if an expression is sufficiently long it's still feasible to exhaust the memory). A classic way to observe this is to use a size function **size< >** (17–23, 56–57) that iterates over a list to obtain a count. What's not always clear is that functors used as default template parameter arguments can be evaluated independently of the template to which they're applied. Take **list_<HEAD, TAIL>::kombine_<N, I_SIZE, O = nop, O_SIZE = 0>** (149–158), if the input size **I_SIZE** is defaulted to **size<>** one might expect it to be calculated, as per a regular function, on invocation. However, as **size<>**'s template parameters are not dependent on **kombine_**'s the compiler instantiates it whenever **list_** is called. This is okay for k-combinations but would unintentionally execute for primes too. To prevent this the default is wrapped by **kombine<N>** (160–161). Alternatively how about having a size attribute i.e. **size = 1** for single elements and **size = HEAD::size + TAIL::size** as they are combined? This is fine when the **HEAD** and **TAIL** are lists. However, they are also used for list expressions; size isn't intrinsic to these and their resultant type is unknown until fully evaluated.

Combinations

The brief is to find all the unique combinations of *k* elements from a set of distinct values. For the purpose of this discussion each element has its own letter and the input set is in alphabetical order.

In Figure 1 (*k*-combinations) the input elements are treated as a queue; at each step the last item is removed from the top of the queue and either placed at the bottom of the output queue when branching right or dropped when going left. If the output reaches the desired length before the input queue is exhausted then it forms a terminal node and branching ceases. Further, if the input is of the required length and the output queue is empty then the input can be directly substituted for the output without further branching.

Here *k*-combinations (Listing 2) are represented with consecutive characters, specified by **c** and **cs**. As previously mentioned **list_<HEAD, TAIL>::kombine<N>** (160–161) wraps the initial call **list_<HEAD, TAIL>::kombine_<N, size< >, nop, 0>**. This takes parameters for the desired combination length, the size of the input list which when first called is the size of the initial list, the output list and its size which are initially empty. The underlying definition

```
…
template< char C, class CS = nil
, char DELIM = '\0' > struct c
: v< char, C, CS, DELIM >            { };
template< char C, char... CS >
struct cs      : c< C, cs< CS... > > { };
template< char C >
struct cs< C > : c< C >              { };

int main( ) {

  /* kombine('abcd', 2)=' ab ac ad bc bd cd'  *
   * powerset('abcd')=' a b c d ab ac ad bc   *
   * bd cd abc abd acd bcd abcd'              */
  cout
   << "\nkombine('abcd', 2)='"
   << cs<'a','b','c','d'>::kombine<2>( )<< "'"
   << "\npowerset('abcd')='"
   << cs<'a','b','c','d'>::powerset( ) << "'";
}
```

Listing 2

`list_<HEAD, TAIL>::kombine_<N, I_SIZE, O, O_SIZE>` (149–158) makes several checks. The first sees if the output can be directly substituted with the input. Otherwise a check is made to see if the output list has reached the desired length. If neither of these are satisfied `list_<HEAD, TAIL >::kkombine_<N, I_SIZE, O, O_SIZE>` (134–147) is called to branch left and right. If the end of the input list is reached i.e. `nil::kombine_<N, size<>, nop, 0>` (36–39) branching ceases with or without the output having reached the specified length. The `powerset` (175–176) is implemented as all the combinations of *k* for 0 to *n* elements.

It can be seen from the enumeration of elements that the permutations, whilst not strictly necessary, are in lexicographic order. Another way of representing combinations is in binary whereby each bit set maps to a corresponding value to print. Investigation of this is left as an additional exercise.

Member template specialisation

You may have noticed that there are a number of member template functors that take an additional, seemingly redundant, parameter; in each case these have a specialisation. For example the power set is all the subsets of an input set including itself and the empty set. In the general case this is lists between the length of the input list `powerset_<N>` to `powerset_<1>` (163–170). A specialisation `powerset_<0>` (172–173) terminates the set with an empty list. Nested classes are dependent on their enclosing template types, hence if explicitly specialised the enclosing classes need to be too. A work-around to this restriction is to provide partial specialisations by adding a dummy parameter.

Sequences

Lists can be long; however, if the content is not arbitrarily distributed then it can be described as a function rather than with individual elements. Evaluating the function and generating the list can then be deferred until first use. This is the case for sieving ranges of numbers with set intervals.

In Listing 3 (Sequences & Sets) a cons of integers `i` is defined for use as a sequence `seq`. The common case produces all integers in ascending order between a low `LO` and high `HI` value. The interval `I` can be altered to increase the step size. Reversing direction by beginning at a high value, ending with a low value and specifying a negative increment will produce a sequence in descending order. At each step the current value is appended to the list and, whilst within bounds, the next step is defined as a sequence of itself with an incremented starting range.

```
...
template< int I, class IS = nil
, char DELIM = ',' > struct i
: v< int, I, IS, DELIM >              { };
template< int I, int... IS >
struct is       : i< I, is< IS... > > { };
template< int I >
struct is< I > : i< I >               { };

template< int LO, int HI, int I = 1
, char DELIM = ',' > struct seq
: conditional<
  ( I > 0 ? LO + I <= HI : LO >= HI - I )
  , i< LO, seq< LO + I, HI, I, DELIM > >
  , i< LO > >::type { };

int main( ) {

  /* seq(-3, 3, 2).uunion(seq(-2, 2, 2))= *
   *     -3,-2,-1,0,1,2,3                 *
   * seq(-3, 3).intrsct(seq(-2, 2, 2))=   *
   *     -2,0,2                           *
   * seq(-3, 3).except(seq(-2, 2, 2))=    *
   *     -3,-1,1,3                        */
  cout
  << "\nseq(-3, 3, 2).uunion(seq(-2, 2, 2))="
  << seq<-3, 3, 2>::uunion< seq<-2, 2, 2>>( )
  << "\nseq(-3, 3).intrsct(seq(-2, 2, 2))="
  << seq<-3, 3>::intrsct<seq< -2, 2, 2>>( )
  << "\nseq(-3, 3).except(seq(-2, 2, 2))="
  << seq<-3, 3>::except<seq<-2, 2, 2>>( );
}
```

Listing 3

Sets

In addition to sequences some set theory is required to sieve. Union, intersection and difference of ascending ordered sets are provided (a description of each follows) although not all will be required. When combining operations it's feasible that earlier operations result in an empty list, hence the member templates for sets (30–34) in `nil`.

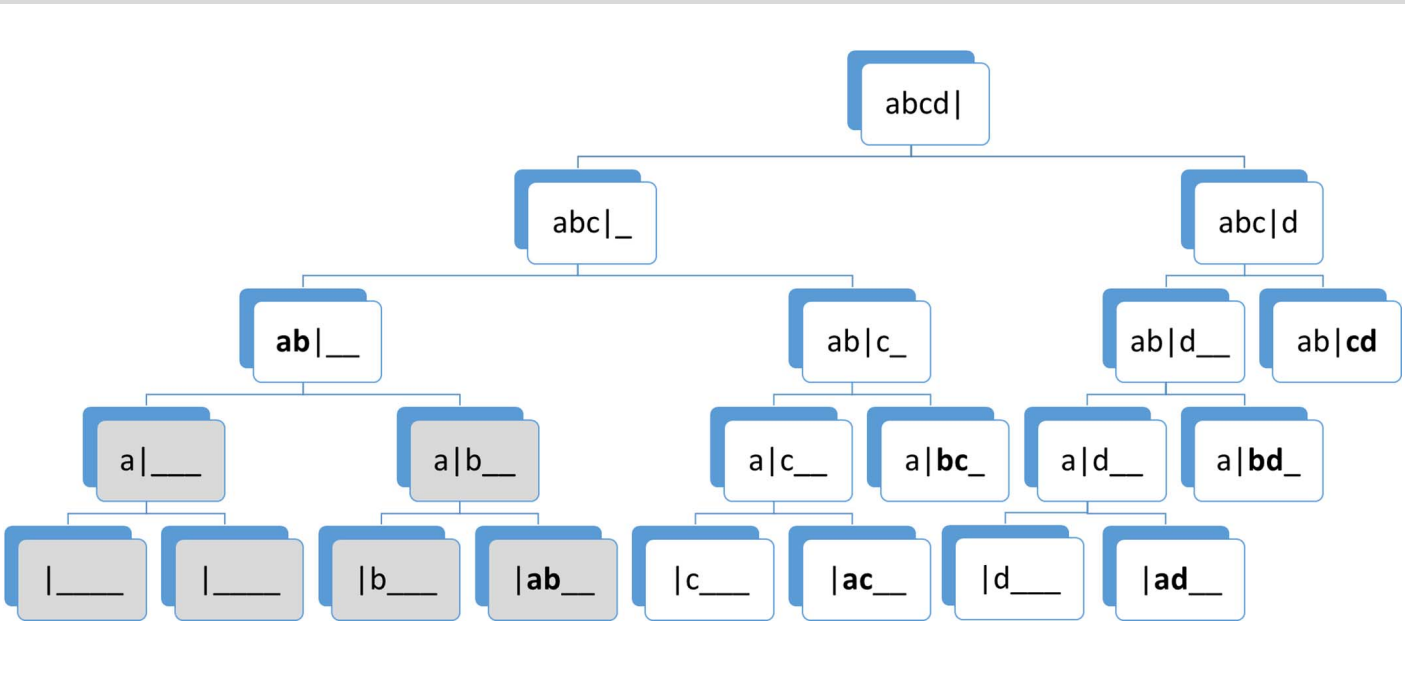


Figure 1

Index (Odd N)	Interval (N + N)	Begin (N x N)	Arithmetic Progression				
3	6	9	9	15	21	27	... 99
5	10	25	... 15	25	35	45	... 95
7	14	49	... 21	35	49	63	... 98
9	18	81	... 27	45	63	81	99

Table 1

Union

The general case `list<HEAD, TAIL>::uunion<RHS>` (80–89) determines which `HEAD`, the `LHS` or `RHS`, is less than the other. Whichever is chosen becomes the `HEAD` of a new list with the rest being the union of its `TAIL` with the other list. The union of a list with an empty list is the list itself (91–92). As it isn't fully defined at the point of parsing its creation is delayed with a concatenation operation i.e. `append_<HEAD, TAIL>`.

Intersection

The general case `list<HEAD, TAIL>::intrsct<R>` (94–109) determines whether the `LHS` and `RHS`'s `HEAD`s are of equal value; if they are one becomes the `HEAD` of the resulting list with the rest comprising the intersection of the respective list's `TAIL`s. Otherwise, the `HEAD` with least value is discarded and the intersection between the remaining elements is performed. Intersection with an empty list (111–112) is, of course, `nil`.

Set difference

Known here as `list<HEAD, TAIL>::except<R>` (114–129) it is the equivalent of minus where any elements in the `LHS` that also appear in the `RHS` are removed. If both the `LHS` and `RHS` are identical then they cancel one another out resulting in the empty list `nil`; otherwise, the `HEAD` of each is compared. If the `LHS` is less than the `RHS` then it becomes the `HEAD` of a new list with the rest being the set difference between its `TAIL` and the `RHS`'s. If it's greater the `RHS`'s `HEAD` is removed and if equal both `HEAD`s are removed with the set difference calculated between the remaining elements. If the `RHS` is an empty list then there is nothing to minus (131–132); as with union the list isn't fully defined at the point of parsing so a concatenation operation i.e. `append_<HEAD, TAIL>` delays its creation.

Sieves

Eratosthenes proposed a simple way for finding primes up to a specified integer using the efficient principal of sieving. There are other well-known variations such as Euler's sieve. The basic mechanism removes multiples of each integer between 2 and *n* thereby leaving only those that are divisible by one and themselves. There are some quick ways to refine this algorithm which also reduce recursive calls. All even numbers with the exception of 2 can be removed from the initial range; that is all multiples seeded from an even index and every other value from those with an odd. Further, as in table 1, higher order progressions have some values that overlap with lower ones. Thus sequences can begin at the square of their seed. Similarly it is unnecessary to go beyond an index that is $\sqrt{n}$ as if *n* is not prime it must have a divisor less than or equal to $\sqrt{n}$ [SICP96]. Table 1 shows the arithmetic progressions required to sieve integers between 2 and 100.

A common solution is to have an array of elements indexed from 1 to *n*, marking 1 for removal, then generating multiples from 2 to the square root of *n*, beginning each sequence with the square of its index and marking these for removal too, and finally printing all unmarked values. Instead the algorithm can be conceived in terms of operations on ordered sets.

In Listing 4 (Eratosthenes Sieve) the template parameter `HI` is the limit value, `N` is the current sieving multiple beginning at 3 and incrementing by 2 for odd intervals. `O` is the output list which will be successively sieved; it begins as a list of a fixed value of 2 followed by the sequence of odd numbers up to and including the limit value. Whilst the square of the multiple i.e. `LO = N * N` is less than or equal to the `HI` boundary every other odd multiple `N + N` is sieved from the output list.

```
...
template< int HI, int N = 3
, class O = i< 2, seq< 3, HI, 2 > >
, int LO = N * N > struct primes
: conditional< LO <= HI
  , primes< HI, N + 2
    , typename O::type::template except<
      seq< LO, HI, N + N > > >
  , O >::type { };

int main( ) {

  /* primes(350)=2,3,5,7,11,13,17,19,23,29,31 *
   * ,37,41,43,47,53,59,61,67,71,73,79,83,89  *
   * ,97,101,103,107,109,113,127,131,137,139  *
   * ,149,151,157,163,167,173,179,181,191,193 *
   * ,197,199,211,223,227,229,233,239,241,251 *
   * ,257,263,269,271,277,281,283,293,307,311 *
   * ,313,317,331,337,347,349                 */
  cout << "\nprimes(350)=" << primes<350>( );
}
```

Listing 4

Summary

In [Weatherhead15] I covered ASCII to integer conversion, roman numerals and palindromes. Each used a variant of the list construct to represent strings. Here it was used to generate combinations with a queue and to sieve sequences by applying set operations. Representative of the way runtime classes are developed the data and methods were encapsulated for reuse. Further ways to experiment with this include implementing other sieves and adding functions to filter and sort. ■

Note

All the code in this article compiles with GCC 4.9.2 and Clang 3.5.2 using *-std=c++11*. However, your mileage may vary with other compilers. Also whilst a null character should not be displayed in a terminal, some platforms show them as a space.

References

[SICP96] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs Second Edition* pp.53–54, 116, 139–145, The MIT Press, 1996.

[Weatherhead15] Nick Weatherhead. Template Programming Compile Time String Functions, *Overload* 128, August 2015.

Further reading

Chris Oldwood. List incomprehension, *CVu* Volume 26 Issue 2 pp.7–8, May 2014.

Stuart Golodetz. Functional Programming Using C++ Templates (Part 1), *Overload* 81, October 2007.

Acknowledgements

I'd like to thank the *Overload* review team for providing their feedback which enabled me to elevate the content presented in this article.

Classdesc: A Reflection System for C++11

C++ lacks direct support for reflection. Russell Standish brings an automated reflection system for C++, Classdesc, up to date.

Classdesc is a system for providing pure C++ automated reflection support for C++ objects that has been in active development and use since the year 2000. Classdesc consists of a simplified C++ parser/code generator along with support libraries implementing a range of reflection tasks, such as serialisation to/from a variety of formats and exposure of the C++ objects to other language environments.

With the increasing adoption of the new C++ standard, C++11, amongst compilers, and in C++ development, it is time for Classdesc to be adapted to support C++11. This is in terms of being able to parse new language constructs, and also to support new concepts, such as enum class. Finally, there is the opportunity to leverage things like variadic templates to generalise and simplify support for function member processing.

Introduction

Classdesc [Madina01] is a system for automated reflection in C++ that has been in continual development since the year 2000, and has been deployed in a number of open source and commercial projects. Classdesc was first published as part of EcoLab 4.D1 in April 2001, before being separated and released as Classdesc 1.D1 in February 2002.

To understand Classdesc, it is necessary to consider one of C++'s more powerful features, the compiler generated constructors and assignment operators. Consider, for example, the default copy constructor, which the C++ compiler generates as needed for any user defined class if the user has not explicitly provided a custom version. To create a new object identical to an existing object, the most common situation is that each component of the existing object is copied to the new object, along with calling the inherited copy constructor of any base class. The copy constructors of each component is called in a hierarchical fashion. At any point in the hierarchy, the programmer can provide a custom copy operator, that suppresses the compiler generated version. An example of a custom copy constructor might be say in the implementation of a vector class, which will typically have a size member, and a pointer to some storage on the heap. The compiler generated copy will copy these members, implementing what is known as a shallow copy. But semantically, one needs a deep copy, where a copy of the data is created, so C++ allows the programmer to provide a deep copy implementation that copies the data as well, possibly even as a 'copy-on-write' implementation that only pays the copy cost when needed.

Just like copy constructors, the default constructor, default destructor and assignment operator are similarly compiler generated to implement a hierarchical call of the respective default constructors/destructors and assignment operators of the component parts. In C++11, two new compiler generated hierarchical methods were added: the move constructor, and move assignment operator, where the object being moved from is left in a valid, but necessarily empty state. Move operations are typically used when the original object is no longer accessed. In the vector example above, the move operation may simply move the pointer to the data into the new structure, rather than copying the data, filling the original pointer in with NULL, a far more efficient operation than regular assignment.

Classdesc takes this notion of compiler generated methods to arbitrary user defined methods, called descriptors, extending the six compiler generated methods defined in C++11. It uses a code generator to create these, based on parsing the C++ class definitions. Because class definitions are closed to extension, descriptors are actually implemented as global functions, with the following signature (eg the pack descriptor):

```
template <class T>
void pack(classdesc::pack_t&,
  const std::string& name, T& a);
```

The Classdesc code generator generates recursive calls of this function on each component of `a`. The first argument is a descriptor dependent structure that can be used for any purpose. In this serialisation example, it will hold the serialised data, or a reference to a stream. The second string argument is used to pass a dot-concatenated name of the members being passed as the third argument – eg `"foo.bar.a"`. This is important for applications that need member names, such as the generation of XML [Bray08] or JSON [ECMA13] plain text representations.

The user has the option of providing their own implementation of the descriptor for specific classes when needed, without having to provide code for the usual (tedious) case. Just as with compiler generated assignment operators, the Classdesc generated code is automatically updated as the class definition evolves during the lifetime of the code. The way this is arranged is to hook the classdesc processor into the build system being used. For example, with a Makefile, one defines an automated rule such as

```
.SUFFIXES: .cd $(SUFFIXES)
.h.cd:
  classdesc -nodef -typeName -i \
  $< json_pack json_unpack >$@
```

Furthermore, if automatic generation of Makefile dependency rules is enabled, then simply adding the line

```
#include "foo.cd"
```

to a C++ source file is sufficient to cause make to invoke classdesc on the header file `foo.h` and create or update the necessary reflection code, and make it available to the compilation module.

The Classdesc distribution comes with descriptors for serialising to/from a binary stream (even machine independent via the XDR library), to/from an XML or JSON streams, and exposure of C++ objects to the JVM via the JNI interface. In EcoLab [Standish03], a C++ simulation environment oriented towards agent based modelling, C++ objects are exposed to a TCL programming environment, providing almost instant scriptability of a C++ written object.

Russell Standish gained a PhD in Theoretical Physics, and has had a long career in computational science and high performance computing. Currently, he operates a consultancy specialising in computational science and HPC, with a range of clients from academia and the private sector. He is also a visiting Senior Fellow at Kingston University in London. Russell can be contacted at hpcoder@hpcoders.com.au

Since the C++ language only exposes limited type information to the programmer, some form of a preprocessing system is required

Alternative C++ reflection systems

In other languages, such as Java, reflection is supported by the system providing objects representing the classes. One can use these class objects to navigate the members and base classes of the objects, querying attributes of those members as you go. Such a reflection system can be called a *runtime* reflection system, as the code navigates the reflected information at runtime.

Classdesc generated code is a static reflection system, in that a descriptor corresponds to a fixed traversal of the object's component tree. This can be used to generate a class object like above, and so formally is at least as powerful as a runtime reflection system. However, for the more usual special purpose applications, such as serialisation, the entire object's serialisation is compiled, leading to better performance than a pure runtime system would.

Since the C++ language only exposes limited type information to the programmer, some form of a preprocessing system is required. Classic special purpose reflection systems include the CORBA IDL [Open12] (which defines another language – the *interface definition language*) which is translated into C++ code to allow exposure of an object's methods to another object running in a remote address space (i.e. remote procedure calls), Qt's moc processor, which extends C++ with a slot and signal model suitable for wiring up GUI components, and Swig [Beazley96] which allows for exposure of object methods and attributes to a range of other programming languages. All of these systems are preprocessors of some language that is sort of an extension of C++, generating standard C++ on its output, so strictly speaking are not reflection systems, but are used for the same sorts of tasks reflection is. Classdesc differs from each of these approaches by processing the same standard C++ header files that the compiler sees.

OpenC++ [Chiba95] tries to generalise this by creating a metaobject processing language which controls a source-to-source code translater, adding in the reflection information before the code is seen by the compiler. Unfortunately, it is now rather dated, and no longer capable of supporting modern dialects of C++.

Chochlík [Chochlík12] reviews a number of other reflection systems, such as SEAL [Roiser04], under the umbrella of 'runtime reflection systems'. SEAL is definitely like that, with a runtime class object being generated by a perl script that analyses the output of *gcc_xml*, a variant of gcc's C++ front end that output an XML representation of the parse tree.

On the other hand, *Mirror* [Chochlík12] is a fully static reflection system, with metaprogramming techniques allowing traversal of type information at compile time. A script *Maureen* uses the *Doxygen* parser to generate the templates supporting the metaprogramming. Unfortunately, the library did not compile on my system, nor did the Maureen script run, but this is perhaps only an indication of the experimental status of a quite ambitious system, rather than unsoundness of the general approach.

The Classdesc system described here was also described by Chochlík as a runtime reflection system, which is not quite accurate. Whilst it is true that Classdesc can be used to generate runtime class objects like runtime reflection systems, so is at least as general as those, that is not how it is usually used. Rather it should be considered as a special purpose static reflection system, admittedly not as general as the full blown static reflection system provided by *Mirror*.

```
class Foo
{
  public:
  virtual string json() const=0;
};
class Bar: public Foo
{
  public:
  string json() const override
    {return ::json(*this);}
  ...
};
```

Listing 1

Dynamic polymorphism

Dynamic polymorphism is the capability of handling a range of different types of object via a common interface, which in C++ is a common base class of the object types being represented, with special methods (called **`virtual`** methods) that reference the specific method implementations appropriate for each specific type.

For Classdesc to work properly, the correct descriptor overload needs to be called for the actual type being represented, which requires that the base class implement a virtual method for calling the actual descriptor. For example, consider an interface **`Foo`** implementing a JSON serialiser (Listing 1).

This ensures that no matter what type a reference to a **`Foo`** object refers to, the correct automatically generated JSON serialiser for that type is called.

Whilst this technique is simple enough, and has the advantage that changes to **`Bar`** are automatically reflected in the json method, it is still tedious to have to provide even the one liner above (particularly if multiple descriptor methods are required), moreover error prone if the base class (**`Foo`** in this case) needs to be concrete for some reason, eliminating the protection provided by the pure virtual specifier.

As an alternative, Classdesc descriptors provide 'mixin' interface classes, that can be used via the CURIOUSLY RECURRING TEMPLATE PATTERN:

```
class Bar: public classdesc::PolyJson<Bar>
{
  ...
};
```

This adds the **`PolyJsonBase`** interface class, which defines the following virtual methods, which covariantly call the appropriate descriptor for the concrete derived class **`Bar`** (Listing 2).

Since the Classdesc parser only parses user defined types, namely classes, structs and enums, language features that only appear within the code bodies of functions or class methods can be ignored

```
struct PolyJsonBase
{
  virtual void json_pack(json_pack_t&,
    const string&) const=0;
  virtual void json_unpack(json_unpack_t&,
    const string&)=0;
  virtual ~PolyJsonBase() {}
};
```

Listing 2

Speaking of **json_unpack**, which is the converse deserialisation operation, the desirable action would be for an object *of the appropriate type* to be created and then populated from the JSON data. To pull off this trick, we have to pack and unpack from/to a smart pointer class, such as **std::shared_ptr** or **std::unique_ptr**. On packing, an extra 'type' attribute is added to the JSON stream, which is used to call a factory create method in the **json_unpack** attribute. The type of the 'type' attribute can be anything, but popular choices are enums (which translate to a symbolic representation within the JSON stream) or strings, using the Classdesc provided **typeName<T>()** method to return a human readable type string for **T**.

```
template <class T>
struct PolyBase: public PolyBaseMarker
{
  typedef T Type;
  virtual Type type() const=0;
  virtual PolyBase* clone() const=0;
  /// cloneT is more user friendly way of getting
  /// clone to return the correct type.
  /// Returns NULL if \a U is invalid
  template <class U>
  U* cloneT() const {
    std::auto_ptr<PolyBase> p(clone());
    U* t=dynamic_cast<U*>(p.get());
    if (t)
      p.release();
    return t;
  }
  virtual ~PolyBase() {}
};
template <class T, class Base>
struct Poly: virtual public Base
{
  /// clone has to return a Poly* to satisfy
  /// covariance
  Poly* clone() const
  {return new T(*static_cast<const T*>(this));}
};
```

Listing 3

```
enum class MyTypes {foo, bar, foobar};
class FooBase: public PolyBase<MyTypes> {};
template <enum class MyTypes T>
class Foo: public Poly<Foo, FooBase>
{
  MyTypes type() const {return T;}
  ...
};
```

Listing 4

Much of the work can be eliminated by adding the following mixin (available in the `polyBase.h` header file) to your polymorphic type, where **T** is the type of your 'type' attribute (Listing 3).

The only thing needed to be implemented in the derived class is the **type()** method. An enum type implementation might be something like Listing 4.

A string type implementation might look like Listing 5 and a full example, including the JSON descriptor methods would be as shown in Listing 6. Note the use of virtual inheritance to ensure that only a single version of PolyJsonBase is in the inheritance hierarchy.

The static method **FooBase::create** needs to be supplied, but even here, Classdesc provides assistance in the form of a **Factory** class (Listing 7).

Parsing of C++11 code

The first task was to test the Classdesc parser/code generator on the new C++11 features. Stroustrup [Stroustrup13, p 1268] provides a convenient 40-point list of the new language features, which provided the starting point for the work to update Classdesc.

Since the Classdesc parser only parses user defined types, namely classes, structs and enums, language features that only appear within the code bodies of functions or class methods can be ignored. A test header file was

```
class FooBase: public PolyBase<std::string> {};
template <class T>
class FooTypeBase: public Poly<T, FooBase>
{
  std::string type() const
  {return classdesc::typeName<T>();}
};
class Foo: public FooTypeBase<Foo>
{
  ...
};
```

Listing 5

C++11 introduces four new containers starting with the name 'unordered', which provide hash map functionality in the standard library

```
class FooBase:
  public classdesc::PolyBase<std::string>,
  public virtual classdesc::PolyJsonBase
{
  static FooBase* create(std::string);
};

template <class T>
class FooTypeBase:
  public classdesc::Poly<T, FooBase>,
  public classdesc::PolyJson<T>
{
  std::string type() const
  {return classdesc::typeName<T>();}
};

class Foo: public FooTypeBase<Foo>
{
  ...
};
```

Listing 6

created with those features that might cause problems to Classdesc, namely:

- enum class (item 7)
- brace style initialisation of members both inline and within constructors (item 1)
- inline member initialisation (item 30)
- alignas type attribute (item 17)

```
class FooBase:
  public classdesc::PolyBase<std::string>,
  public classdesc::Factory<FooBase, std::string>
{};

template <class T>
class FooTypeBase:
  public Poly<T, FooBase>,
  public PolyJsonBase<T>
{
  std::string type() const
  {return classdesc::typeName<T>();}
};

class Foo: public FooTypeBase<Foo>
{
  Foo() {registerType(type());}
};
```

Listing 7

- constexpr (item 4)
- default and delete declarations (item 31)
- spaceless closing of nested templates (item 11)
- attributes, in particular [[noreturn]] (item 24)
- noexcept attribute (item 25)
- deduced method return type (item 23)
- variadic template arguments (item 13)

and then a set of unit test cases linking this header file, and all the classdesc provided descriptors was created to ensure completeness of the work.

Of these new features listed above, about half required changes to Classdesc, which will be described in more detail in the following section.

New C++11 features

Enum class

C++ introduces a new user defined type called enum class. This pretty much fixes a name scope problem with the original C enum type. Classdesc was modified to process enum classes in the same way that it processes enums, which includes the generation of a symbolic lookup table (map of the enum tag names as strings to/from the numerical tag values). This is desirable for generation and parsing of formats such as XML or JSON, for which the numerical values are not meaningful, and may even differ from the C++ numerical values if processed by a different language.

Smart pointers

C++11 adds a couple of new smart pointers: the **`unique_ptr`** and **`shared_ptr`**, and deprecates an existing one (**`auto_ptr`**). These pointers, in particular the **`shared_ptr`**, have been available in external libraries for some time, notably in the boost library [Boost], and then later in a precursor to the C++11 standard library known as TR1 [TR05]. Classdesc has required the use of **`shared_ptr`**s for some time to adequately support dynamic polymorphism, as well as containers of noncopiable objects. Since the design of Classdesc is to not rely on 3rd party libraries like boost, by preference it will use C++11 **`std::shared_ptr`** if available, otherwise the TR1 shared pointer, and only use boost shared pointers as a last resort.

The problem is that these three different smart pointer implementations are distinct types. The solution in Classdesc is to define a typedef alias of **`shared_ptr`** in the classdesc namespace, which refers to whichever implementation is being used. A similar consideration applies to a variety of metaprogramming support functions (eg **`is_class`**) which have been introduced into the language via external libraries before being finally standardised in C++11.

If a pre-C++ compiler is used, one can select the version of **`shared_ptr`** to be used by defining the **`TR1`** or the **`BOOST_TR1`** macros respectively. If neither macro is defined, then the **`tr1`** namespace is assumed to be defined in the standard **`<memory>`** header file, as is the case with Microsoft's Visual C++.

Classdesc now introduces metaprogramming type attributes to represent the sequence and associative container concepts from the Standard Template Library (STL)

New STL containers

C++11 introduces four new containers starting with the name 'unordered', which provide hash map functionality in the standard library, along with a single linked list. Whilst, one could have extended the standard Classdesc provided descriptors to cover these new containers, it highlighted that it was high time to deal with these in a more generic way. Consequently, Classdesc now introduces metaprogramming type attributes to represent the sequence and associative container concepts from the Standard Template Library (STL). Descriptors are now implemented in terms of these type attributes, instead of referring directly to the container types: vector, list, set etc. The STL container types have these attributes defined in the `classdesc.h` header. Users can avail themselves of this descriptor support for their own custom container types merely by defining an **`is_sequence`** or an **`is_associative_container`** type attribute as appropriate:

```
namespace classdesc
{
  template <>
  struct is_sequence<MySequenceContainer>
  { static const bool value=true; }
};
```

New fundamental types

C++11 adds a whole slew of new types, including explicitly 16 and 32 bit wide characters (**`char16_t`** and **`char32_t`**), a long long integer and a number of type-name aliases for explicitly referring an integer's size (eg **`int64_t`**). Typename aliases do not cause a problem for classdesc, as their use will be covered by the type they are an alias for. However, traditional descriptor implementations required explicit implementations for all the fundamental types, so supporting these new types requires a lot of additional code. So the decision was made to rewrite as much descriptor code as possible using type traits such as **`is_fundamental`**, and use metaprogramming techniques [Veldhuizen95].

There is, however, one place where all the basic types need to be enumerated, and that is in the implementation of the typeName template, which returns a human readable string representation of the type. Explicit template specialisation for each fundamental type needs to be provided. One might ask why **`type_info::name()`** could not be used for this purpose. Unfortunately, the standard does not specify how compiler should map type names to strings, and compilers often choose quite mangled names that are unsuitable for some reflection purposes, such as in XML processing, in a compiler independent way.

Opportunities

Move operators

Certain Classdesc types, such as pack buffers, are unable to be copied. In particular, lacking copiability restricts the use of these objects in containers, unless stored as smart pointers. C++11 provides the concept of move construction and assignment, where the source object in a valid, but empty state, which can usually be implemented in an inexpensive and simple fashion. An object with a move operator can be used in a C++11 container. The update to Classdesc provides implementations of these move operators where appropriate to extend their use cases.

Functional support

Classdesc provides a metaprogramming library to support the analysis of method signatures for supporting the exposure of methods to other programming environments, such as the JVM. Key metaprogramming requirements are arity (number of arguments), the individual types of each argument **`Arg<F, i>::type`** and the return type of the function or method type **`F`**. For pre-C++11 compilers, this was implemented for all cases up to a certain number of arguments (usually 10), with the code being generated by a Bourne shell script. This peculiar break from a pure C++ solution was deemed a necessary evil – in practice the provided code limited to 10 arguments suffices for most practical cases, and if not, then access to a Bourne shell to generate support for higher arities is usually not difficult to arrange.

Nevertheless, C++11's variadic templates allows the possibility of handling an arbitrary number of function arguments without the need to generate specific templates from a script. At the time of writing, though, this has not been implemented in Classdesc.

A proposal for an extension to the C++ language

Since Classdesc-provided reflection naturally maps to the same recursive hierarchical concept as do the compiler generated constructors and assignment operators, and C++11 has introduced a new syntactic construct based on the default keyword that forces the compiler to generate these methods, this leads to a natural proposal. That is, allow any method signature suffixed by default to have a compiler generated body that recursively applies that method signature to the object's components.

For example, consider a struct declared as:

```
struct Example
{
  A a;
  B b;
  void pack(pack_t& buf)=default;
};
```

The compiler is instructed to generate the method:

```
void Example::pack(pack_t& buf)
{
  a.pack(buf);
  b.pack(buf);
}
```

There are some subtleties that need to be worked out. For example, if the class is a mix of private and public attributes, we need to be able to specify whether private attributes should be processed (eg in serialisation applications) or not (eg in exposing object APIs to another language), a feature currently implemented as a flag on the classdesc code generator

command line. One suggestion is to qualify the default keyword with public/protected/private to indicate which attributes are processed:

```
void pack(pack_t& buf)=default private;
```

The question is what do if the access qualifier is not specified. Should it default to private, which is effectively what the existing compiler generated methods do, or should it be public, which would be the more common usage.

The next issue is how to handle the situation where a type does not have a method of that name defined? The obvious solution is to borrow from operator overloading, and if (say) `a.pack(buf)` is an invalid expression, substitute `pack(a,buf)`, resolved according to the usual namespace resolution rules. This will allow writers of descriptors to add new descriptors to existing types, particularly the fundamental types.

The final issue to address is how to implement an equivalent of the covariant member name feature of Classdesc, which is available as a string passed as the second argument of the descriptor. The most obvious suggestion is a magic type declared in namespace std (in the same way that `std::initializer_list` is magically populated by brace initialisers), that will be populated by the appropriate hierarchical list of names of the member.

```
void xml_pack(pack_t& buf,
  std::refl_name)=default;
```

This will be populated in the compiler generated code as follows:

```
void Example::xml_pack(pack_t& buf,
  std::refl_name nm)
{
  a.xml_pack(buf,nm+"a");
  b.pack(buf,nm+"b");
}
```

`refl_name` will be a sequence, and can be iterated over by the usual means, with perhaps a concatenation method to return the dot separated list currently used by Classdesc.

Proposals for reflection in C++17

Support for reflection in standard C++ has been discussed a number of times, but generally been dismissed as requiring metaobjects to be present in the resulting executable, even though the classes themselves may not end up being used by the programmer, and hence can be optimised away. Clearly, this is only an objection for a traditional runtime reflection systems, not static systems – for example, even using Classdesc to generate metaobjects, the metaobject will only exist if explicitly created by the programmer calling the appropriate descriptor, which is just a standard function that can be eliminated by the linker if not used.

Nevertheless, Carruth and Snyder [Carruth13] issued a general call for reflection proposals for consideration of inclusion in the next (C++17) standard. To date, two proposals have been put forward: Chochlík's [Chochlík15], which is largely based on the *Mirror* library and Silva and Auresco's [Auresco15], who propose extending the keywords typename and typedef to return variadic templates that can be used in a metaprogramming context, but does not specify any extensions to the standard type traits library. In Chochlík's proposal, a new operator (tentatively `mirrored`) returns a compile time static metaobject, that can iterated over, or otherwise queried. In a way, the two proposals mesh together quite well. The mirror library is a quite well thought out reflection library extending `std::type_trait`s, but the actual structure of the MetaObjectSequence is not well specified in Chochlík's proposal. On the other hand, Silva and Auresco's idea of using variadic templates to encode the MetaObjectSequence concept at least fits in with how 'loops' are currently implemented in C++11 metaprogramming, and has the further advantage of not requiring new keywords.

In either proposal, Classdesc-like functionality could be achieved by arranging for the generic descriptor template to be a metaprogrammed loop of the class members.

Conclusion

Classdesc has been under development for 15 years, and has a reputation for being a solid, no-fuss, portable reflection system for C++. With the increasing use of C++11 code, it was time to bring Classdesc up to date with the new C++ standard, which has now been achieved. ■

References

[Auresco15] Daniel Auresco, Cleiton Santoia Silva. 'From a type T, gather members name and type information, via variadic template expansion. Technical Report N4447, ISO/IEC JTC, 2015. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4447.pdf

[Beazley96] David M. Beazley. 'SWIG : An easy to use tool for integrating scripting languages with C and C+'+. *In Proceedings of 4th Annual USENIX Tcl/Tk Workshop*. USENIX, 1996. http://www.usenix.org/publications/library/proceedings/tcl96/beazley.html

[Boost] Boost C++ Libraries. http://www.boost.org/

[Bray08] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and François Yergeau. 'Extensible markup language (XML) 1.0 (fifth edition)'. Technical report, W3C, 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[Carruth13] C. Carruth, J. Snyder. 'Call for compile-time reflection proposals'. Technical Report N3814, ISO/IEC JTC, 2013. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3814.html

[Chiba95] Shigeru Chiba. 'A metaobject protocol for C++'. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299, 1995.

[Chochlík12] Matúš Chochlík. 'Portable reflection for C++ with mirror'. *Journal of Information and Organizational Sciences*, 36(1):13.26, 2012.

[Chochlík15] Matúš Chochlík. 'Static reflection'. Technical Report N4451, ISO/IEC JTC, 2015. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4451.pdf.

[ECMA13] ECMA. The JSON data interchange format. Technical Report ECMA-404, ECMA International, 2013. http://www.ecma-international.org/publications/standards/Ecma-404.htm

[Madina01] Duraid Madina and Russell K. Standish. 'A system for reflection in C++'. In *Proceedings of AUUG2001: Always on and Everywhere*, page 207. Australian Unix Users Group, 2001.

[Open12] Open Management Group. 'C++ language mapping'. Technical report, OpenManagement Group, http://www.omg.org/spec/CPP/1.3, 2012. Version 1.3.

[Roiser04] S. Roiser and P. Mato. 'The SEAL C++ reflection system'. In *Proceedings of Computing in High Energy Physics*, CHEP '04, Interlaken, Switzerland, 2004. http://chep2004.web.cern.ch/chep2004/

[Standish03] Russell K. Standish and Richard Leow. 'EcoLab: Agent based modeling for C++ programmers'. In *Proceedings SwarmFest 2003*, 2003. arXiv:cs.MA/0401026.

[Stroustrup13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, 2013.

[TR05] Draft technical report on C++ library extensions. *Technical Report DTR 19768*, International Standards Organization, 2005.

[Veldhuizen95] Todd Veldhuizen. Using C++ template metaprograms. C++ Report, 7:36.43, 1995.

QM Bites : Maximising Discoverability of Virtual Methods

C++11 introduced override as a contextual keyword. Matthew Wilson encourages us to use it.

TL;DR:

Reduce opacity in C++ inheritance hierarchies w **`override`** k/w (or comments in C++03/earlier)

Bite:

One of the myriad sources of confusion about C++'s ambiguous syntax is in determining whether or not a virtual function is one prescribed in the type one is currently examining or prescribed from a parent class. Consider the following:

```
class MidLevel
  : public TopLevel
{
  . . .
  virtual void SomeMethod();
};
```

There are several possible interpretations:

1. The virtual method **`SomeMethod()`** is defined and implemented only within the class **`MidLevel`** (and may be overridden by derived types);?
2. The virtual method **`SomeMethod()`** is defined by the class **`TopLevel`** (or one of its ancestors) in which it is pure, so **`MidLevel::SomeMethod()`** is (at this level) its one and only definition;?
3. The virtual method **`SomeMethod()`** is defined and implemented by the class **`TopLevel`** (or one of its ancestors) so **`MidLevel::SomeMethod()`** is an override (which may or may not invoke **`TopLevel::SomeMethod()`** in its implementation);?

The only way to know is to examine the definition of the class **`TopLevel`**, or the definition of (a) parent class of **`TopLevel`**, or the definition of (a) grandparent class of **`TopLevel`**, or …

Application of the C++ 11 keyword override is a boon to discoverability, in that it connotes that the method is an override of a method declared/defined by a parent class. Hence, Listing 1 implies either case 2 or 3 above.

```
class MidLevel
 : public TopLevel
{
  . . .
  virtual void SomeMethod() override;
};
```

Listing 1

Matthew Wilson Matthew is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au.

```
class MidLevel
 : public TopLevel
{
  . . .
  virtual void SomeMethod() /* override */;
};
```

Listing 2

With C++-98/03 (or any compiler that does not support C++ 11's override), an alternative declarative technique is simply to use a comment, as in Listing 2, or object-like pseudo-keyword macro (that resolves to **`override`** with compilers that support the keyword, and to nothing with those that do not), as in Listing 3 and Listing 4.

Of course, the virtue of the new keyword is far greater than that of connotation of design intent – it facilitates *enforcement* of design intent. If Listing 1 is presented to the compiler in case 1, it will be a compiler error, since one cannot override something that does not (previously) exist. That's great.

Just as importantly, it guards against coding errors and/or design changes, in requiring that the overriding method matches the overridden method. If Listing 1 compiles, but then the method signature (or return type, or cv-qualification, or universality) changes in the parent class – the *fragile base class* problem – it will be a compile error. That's great too.

(Obviously, neither the comment-form nor the macro-form do any enforcement with non-C++-11-compatible compilation, but it still is a non-trivial amount of help for the human side of things.)

Prior to the advent of **`override`** my practice was to distinguish between case 1 and cases 2/3 by placing the **`virtual`** keyword in comments, as in Listing 5.

Should you wish, you may do this too, but since **`override`** does a superior job in connoting overriding, you might prefer to elide it completely, as in Listing 6.

```
#ifdef
  ACMESOFT_COMPILER_SUPPORTS_override_KEYWORD
# define ACMESOFT_override_K    override
#else
# define ACMESOFT_override_K
#endif
```

Listing 3

```
class MidLevel
 : public TopLevel
{
  . . .
  virtual void SomeMethod() ACMESOFT_override_K;
};
```

Listing 4

One of the myriad sources of confusion ... whether or not a virtual function is one prescribed in the type one is currently examining

```
class MidLevel
 : public TopLevel
{
  . . .
  /* virtual */
  void SomeMethod() ACMESOFT_override_K;
};
```

Listing 5

```
class MidLevel
 : public TopLevel
{
  . . .
  void SomeMethod() ACMESOFT_override_K;
};
```

Listing 6

After all this you might be wondering what we do about distinguishing between cases 2 and 3. That's another story (but if I give you a hint and suggest that case 3 is pretty much a design smell, pertaining to ***modularity*** as well as ***discoverability***, you might get there before we visit that subject in this place). ■

And the winners are...

In the last *Overload* we invited our readers to vote for their favourite articles in 2015 from *CVu*, which is our sibling magazine for members, and *Overload*.

For *Overload*, in joint first place we have:

Adam Tornhill, Meet the Social Side of Your Codebase

Andrew Sutton, Introducing Concepts

Peter Sommerlad, Variadic and Variable Templates

For *CVu*, we have a clear winner with over 50% of the votes:

Roger Orr, One Definition Rule

In joint second place we have

Adam Tornhill, Writing a Technical Book

Richard Falconer, Functional Programming in C++

Thank you for everyone who took time to vote, and for those who wrote. We can't offer a prize to these winners, just the mention here. A huge number of other writers got a vote – so be assured if you wrote for us someone probably thoroughly enjoyed what you had to say. Keep up the good work.

The article titles above link to the articles if you are reading this as a PDF. *Overload* articles are publicly available, but you must be a member (and logged in) to access the *CVu* ones. If you're not a member yet, why not join?

Join ACCU

So Why is Spock Such a Big Deal?

Spock testing in a Java environment is all the rage. Russel Winder talks through the history of testing on the JVM and demonstrates why Spock is so groovy.

We will take as read the fact that all programmers know that testing is a good thing, and that they do it as a matter of course. Whether programmers use test-driven development (TDD), behaviour-driven development (BDD), and/or some other process, we will take as read that all programmers have good test coverage of their systems at all times.

So the only question is which testing framework?

For some platforms, there is very little argument about which test framework to use. For example, with Go and D there is a built-in framework that most people use. There tend to be extensions, but most programmers just use what is provided by default. With C++ though, as most ACCU members will know, there has been a long history of a plethora of frameworks. This article is though about the Java Platform. If you think this just means Java and JUnit, then… wrong.

A bit of history

Given that we are talking about the Java Platform milieu only here…

In the beginning (mid-1990s), Java was used for making Web browser plugins, and few plugin writers really cared much about testing. As Java started being used server-side, the development process known as TDD jumped from its Smalltalk roots to the rapidly expanding Java-verse. Kent Beck and Erich Gamma created JUnit based on the architecture of sUnit that Kent Beck had developed for use with Smalltalk in the early 1990s. Unlike other programming languages of the time, where the model was 'everyone writes their own test framework', the model in the Java-verse rapidly became 'use JUnit'. JUnit just became an integral part of the Java-verse, treated almost as a part of the Java Platform – even though it was, and is, not.

Around 2003 though there was a stirring in the Java-verse: generics and annotations were coming to Java 5. Although this wasn't a change of computational model, and so JUnit could work as it had ever done, annotations brought a whole new way of thinking about Java code and about test frameworks. Cédric Beust saw this as an opportunity and set about creating TestNG. This replaced the naming conventions and use of inheritance that was integral to the JUnit way of working, with the use of annotations, and it changed the way programmers implemented their tests.

JUnit remained in maintenance mode whilst TestNG rushed into the new Java 5 style of programming. Now there were two. However TestNG, the 'new kid in town' was having to fight the incumbent and entrenched JUnit for usage. For many, working with Java meant using Eclipse and JUnit came as standard, whereas TestNG was an 'added extra'. Programmers had to do something to switch from JUnit to TestNG, and generally didn't, even though TestNG brought new capabilities as well as a Java 5 way of working.

After what seemed like an age after the release of Java 5, a new JUnit appeared, JUnit4 – the old JUnit was relabelled JUnit3. In many, many ways, JUnit4 was just a copy of TestNG. Where JUnit4 and TestNG differ, TestNG is generally the better framework. The most obvious 'grand difference' was that JUnit was a unit testing framework whereas TestNG was a test framework covering unit, integration, smoke, and some system testing. However JUnit4 was JUnit and therefore, at least in many Java programmers' minds, the one true Java test framework. More importantly though JUnit4 was seen as an upgrade from JUnit3 and so it was very easy for all the IDEs, and in particular Eclipse, to claim they were modern and hip with Java 5 by switching JUnit3 out and JUnit4 in.

Where else have we seen technical superiority ignored in the market?

So JUnit4 became established as the Java 5 style test framework, leaving TestNG as a minor player. Of course, some people didn't bother to move from JUnit3 since they could see no benefit to the use of annotations rather than a naming convention and use of inheritance. These people argued that the switch from a naming convention to use of annotations didn't actually bring any new capability: JUnit4 was not actually a functional improvement over JUnit3. TestNG brought integration and system testing mindsets yes, but most programmers still thought testing meant unit testing. The JUnit3 'die hards' were, indeed are, wrong.

Things get groovy

Concurrent with the JUnit4 vs. TestNG vs. JUnit3 battle came the invention, development and rise of the Groovy programming language.[1] Whereas Java is a statically-typed compiled language, Groovy is a dynamic language. Yes, Groovy is compiled to JVM bytecodes just as Python is compiled to PVM bytecodes, but a Groovy program is a dynamically typed system. Perhaps bizarrely, Groovy now has the capability of being statically type checked, and indeed fully statically compiled. This makes it a competitor to Java as a statically typed language as well as being a dynamic symbiote to the static Java.

In its dynamic language guise, Groovy is much closer to Smalltalk than Java ever can be. Algorithms, programming techniques, and idioms of Smalltalk are much easier to represent in Groovy than they are in Java. The JUnit3 way of working is completely natural in Groovy where it can be a little awkward in Java. Of course Groovy can work with JUnit4 and TestNG since it is symbiotic with Java, inter-working very easily with Java.

For a while Groovy used principally the JUnit3 approach, to the extent of integrating it directly into the runtime system via the `GroovyTestCase` class. Of course, JUnit4 and TestNG could be used, but Groovy arose in

Russel Winder Ex-theoretical physicist, ex-UNIX system programmer, ex-academic. Now an independent consultant, analyst, author, expert witness and trainer. Also doing startups. Interested in all things parallel and concurrent. And build. Actively involved with Groovy, GPars, GroovyFX, SCons, and Gant. Also Gradle, Ceylon, Kotlin, D and bit of Rust. And lots of Python especially Python-CSP.

1. Groovy is now (2015 Q4) a top-level Apache project, and is properly called Apache Groovy.

Where else have we seen technical superiority ignored in the market?

a fundamentally JUnit3 context, and the model of working fitted very well.

Then around 2007–2009 Abstract Syntax Tree Transformations (AST Transforms) came to Groovy, formally released in Groovy 1.6. These are very similar to what are called macros in other languages or annotations in Java. Many see them as ways of providing just the sort of thing that annotations and macros can provide. This misses some of the truly devious things that can be achieved with Groovy AST transforms. Peter Niederwieser, however, did not miss the potential: he proceeded to create Spock, based on the new AST transform capabilities with a desire to escape the straight-jackets that are JUnit3, JUnit4, and TestNG.

Testing by example

Clearly the best way of showing Spock's, indeed any test framework's, capabilities and comparing with other frameworks, e.g. JUnit3, JUnit4, and TestNG, is by example. For this we need some code that needs testing: code that is small enough to fit on the pages of this august journal,[2] but which highlights some critical features of the test frameworks.

We need an example that requires testing, but that gets out of the way of the testing code because it is so trivial.

We need factorial.

Factorial is a classic example usually of the imperative vs. functional way of programming, and so is beloved of teachers of first year undergraduate programming courses.[3] I like this example though because it allows investigating techniques of testing, and allows comparison of test frameworks.

Factorial is usually presented via the recurrence relation:

$$f_0 = 1$$

$$f_n = nf_{n-1}$$

However, this way of presenting the semantics of factorial is just the beginning of the tragedy that is most people's first recursive (functional) implementation.

Being naïvely imperative

Let us completely avoid the whole recursive function thing for this article, since we are focusing on testing.[4] Let us instead consider what almost every programmer would write as an iterative (imperative) implementation using Java[5] see Listing 1.

We can imagine a programmer constructing the JUnit4 test as shown in Listing 2 and feeling very pleased with themselves.

2. This is not though an annual magazine sent out only in August.
3. Though with the changes to UK school curriculum in 2014 of IT to computing, this example may well have to move down the age scale.
4. I shall reserve the right to rant about this in another article.
5. We will ignore the whole 'Integer' vs. 'int' thing for the purposes of this article, which is about testing not benchmarking.

```
package uk.org.russel.stuff;

  public class Factorial_Naïve {
    public final static Integer iterative(
        final Integer n) {
      Integer total = 1;
      for (Integer i = 2; i <= n; ++i) {
        total *= i;
      }
      return total;
    }
  }
```

Listing 1

There is so much wrong with these codes, it is difficult to know where to start – and switching to TestNG with this style of testing will not help. The two most obvious problems with this test are:

1. What happens for negative arguments? (Factorial is undefined for negative arguments.)
2. What happens for arguments greater than 13? (The above implementation will give the wrong answer.)

Point 1 is just highlighting the fact that most programmers tend to consider testing only the success modes of their code and fail to deal with the failure modes. Good QA people tend to immediately break things, exactly because they look at failure modes, which leads to tensions, sometimes animosity, but is a road that eventually leads to DevOps.

Point 2 is actually also about failure modes but is about underlying implementations rather than testing of the domain of the units. In this case

```
package uk.org.russel.stuff;

import org.junit.Test;
import static
    org.junit.Assert.assertEquals;

import static
    uk.org.russel.stuff.Factorial_Naïve.iterative;

public class Test_Factorial_Naïve_JUnit4_Java {
  @Test  public void zero() {
    assertEquals(new Integer(1), iterative(0));
  }
  @Test  public void one() {
    assertEquals(new Integer(1), iterative(1));
  }
  @Test  public void seven() {
    assertEquals(new Integer(5040), iterative(7));
  }
}
```

Listing 2

most programmers tend to consider testing only the success modes of their code and fail to deal with the failure modes

it is about the fixed size of JVM integral types and the overflow that occurs. This means we must immediately give up using `Integer` and switch to `BigInteger`, how else can we deal with a function whose values are generally ####### big numbers.[6]

```
package uk.org.russel.stuff;

import java.math.BigInteger;

public class Factorial {

  public final static BigInteger iterative(
      final Integer n) {
    if (n < 0) {
      throw new IllegalArgumentException(
"Argument must be a non-negative Integer.");
    }
    return iterative(BigInteger.valueOf(n));
  }

  public final static BigInteger iterative(
      final Long n) {
    if (n < 0l) {
      throw new IllegalArgumentException(
"Argument must be a non-negative Long.");
    }
    return iterative(BigInteger.valueOf(n));
  }

  public final static BigInteger iterative(
      final BigInteger n) {
    if (n.compareTo(BigInteger.ZERO) < 0) {
      throw new IllegalArgumentException(
"Argument must be a non-negative BigInteger.");
    }
    BigInteger total = BigInteger.ONE;
    if (n.compareTo(BigInteger.ONE)  > 0) {
      BigInteger i = BigInteger.ONE;
      while (i.compareTo(n) <= 0) {
        total = total.multiply(i);
        i = i.add(BigInteger.ONE);
      }
    }
    return total;
  }

}
```

Listing 3

6. Note that switching from 'Integer' to 'Long' serves no useful purpose other than raising the point of failure from arguments greater than 13 to arguments greater than 20.

Becoming less naïve

Listing 3 is something of a transliteration of the earlier implementation to using `BigInteger`. Overloading is employed to provide implementations for different argument types to try and fully cover the domain. Note that negative arguments are now dealt with.

I think we can all agree that writing code in Java working with `BigInteger` is somewhat less than pleasant.

```
package uk.org.russel.stuff;

import java.math.BigInteger;

public class Factorial {

  public final static BigInteger iterative(
      final Integer n) {
    if (n < 0) {
      throw new IllegalArgumentException(
"Argument must be a non-negative Integer.");
    }
    return iterative(BigInteger.valueOf(n));
  }

  public final static BigInteger iterative(
      final Long n) {
    if (n < 0l) {
      throw new IllegalArgumentException(
"Argument must be a non-negative Long.");
    }
    return iterative(BigInteger.valueOf(n));
  }

  public final static BigInteger iterative(
      final BigInteger n) {
    if (n.compareTo(BigInteger.ZERO) < 0) {
      throw new IllegalArgumentException(
"Argument must be a non-negative BigInteger.");
    }
    BigInteger total = BigInteger.ONE;
    if (n.compareTo(BigInteger.ONE)  > 0) {
      BigInteger i = BigInteger.ONE;
      while (i.compareTo(n) <= 0) {
        total = total.multiply(i);
        i = i.add(BigInteger.ONE);
      }
    }
    return total;
  }

}
```

Listing 4

Have we lost anything by not using annotations to specify test methods? Not really.

```
package uk.org.russel.stuff;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

import java.math.BigInteger;

import static
  uk.org.russel.stuff.Factorial.iterative;

public class Test_Factorial_JUnit4_Java {

  @Test
  public void zero() {
    assertEquals(BigInteger.ONE, iterative(0)); }

  @Test
  public void one() {
    assertEquals(BigInteger.ONE, iterative(1)); }

  @Test
  public void seven() {
    assertEquals(BigInteger.valueOf(5040),
                 iterative(7)); }

  @Test(expected=IllegalArgumentException.class)
  public void minusOne() { iterative(-1); }

}
```

Listing 5

Of course we must have some tests (see Listing 4). Which I guess is fine, well fine-ish, anyway.

Being Groovy

Instead of using Java for the test code, we can use Groovy code. Although Groovy is a dynamic language whereas Java is a statically typed one, Groovy is based on the exact same data model and so we can just access the JUnit4 features directly, as shown in Listing 5.

One could argue that there is little or no benefit accruing here to using Groovy rather than Java, even though being able to render the `BigInteger` literals more readably makes for a nicer read of the testing code. And there are no semicolons.

As previously there is little or no benefit to using TestNG compared to JUnit4 in this situation.

So why use Groovy at all?

Two obvious reasons spring to mind:

1. We can rewrite the Factorial implementation in Groovy: Groovy can be a very nice, statically-typed, compiled language simply by using the `@CompileStatic` AST transform. We could write the factorial implementations using Groovy code as shown in Listing 6, which produces the same results at fundamentally the same performance of the earlier Java code. Having `BigInteger` literals and the ability to define operators on types,[7] the code is much easier to read and much easier to maintain. Despite this superiority of Groovy, many think they have to use Java for production code.
2. We can use Spock.

```
package uk.org.russel.stuff

import groovy.transform.CompileStatic

@CompileStatic
class Factorial_Groovy {

  static BigInteger iterative(Integer n) {
    if (n < 0) {
      throw new IllegalArgumentException(
'Argument must be a non-negative Integer.')
    }
    iterative(n as BigInteger)
  }

  static BigInteger iterative(Long n) {
    if (n < 0) {
      throw new IllegalArgumentException(
        'Argument must be a non-negative Long.')
    }
    iterative(n as BigInteger)
  }

  static BigInteger iterative(BigInteger n) {
    if (n < 0G) {
      throw new IllegalArgumentException(
'Argument must be a non-negative BigInteger.')
    }
    def total = 1G
    if (n > 1G) { (2G..n).forEach{total *= it} }
    total
  }

}
```

Listing 6

Enter Spock

Let's dive straight into an example: Listing 7 is a Spock version of the tests of the Java implementation of factorial.

7. We leave for another article a rant about how excluding operator definition from Java may not actually have been as a good a programming language design choice as the Project Green people thought in the early 1990s.

The idea of writing one test method for each test case is fine in principle. Actually it is a very, very good idea. However the idea of manually writing one test method for each test case is clearly a very, very silly one.

```
package uk.org.russel.stuff

import spock.lang.Specification

import static
  uk.org.russel.stuff.Factorial.iterative

class Test_Factorial_Spock_Groovy
  extends Specification {

  def zero() {
    expect:
    iterative(0) == 1G
  }

  def one() {
    expect:
    iterative(1) == 1G
  }

  def seven() {
    expect:
    iterative(7) == 5040G
  }

  def minusOne() {
    when:
    iterative(-1)
    then:
    thrown(IllegalArgumentException)
  }
}
```

Listing 7

Very Groovy. And very much a return to the sUnit/JUnit3 sort of thinking in that inheritance is used to deal with marking classes that are test code, and method names are important: Spock assumes all except some specific method names are test methods, feature method in Spock nomenclature. Have we lost anything by not using annotations to specify test methods? Not really. Have we gained anything using Groovy? Apart from a much nicer way of expressing `BigInteger` literals, arguably not – except that we can use Spock. Has Spock brought something to the case. Definitely. The whole naming and structuring of tests is revolutionized. Assuming methods are test methods cleans things up, but the real win is that Spock steps away from the traditional test method structure. Instead, Spock uses a block structuring of test methods to give much more of an obvious Arrange–Act–Assert structure. Labels introduce blocks of code. Expect blocks are sequences of Boolean expressions that are assertions about the state – a mix of act and assert. when/then block pairs provide an action separate from the assertion. In this last case we are seeing the Spock way of specifying that an exception is expected. If nothing else the code reads much more easily and enables both TDD- and BDD-style thinking about tests.

Of course, there is a lot more to Spock that makes it the framework of choice for Java and Groovy codebases – also possibly Scala, Ceylon, and Kotlin codebases. Let us delve into arguably the most important.

Getting parameterized: the Spock variant

The idea of writing one test method for each test case is fine in principle. Actually it is a very, very good idea. However the idea of manually writing one test method for each test case is clearly a very, very silly one. We should be getting the framework to write the methods for us given input of a table of test cases. Data-driven testing is a very good idea, and any framework that does not support this cleanly, with easy use, is clearly not fit for purpose.

TestNG has 'data providers' which work very well. JUnit has 'parameterized tests' which are a little less nice than TestNG data providers but can achieve more or less the same thing.[8]

Listing 8 is an extended version of a test for the Java iterative factorial implementation, using some of the power of Spock. Here we can see the power associated with use of Groovy:

- Method names can be arbitrary strings.
- Operator definition allows a nice syntax for internal DSLs giving:
  - a tabular structure of data (as in the first test method); and
  - providing an iterable over which to iterate (as in the second method).

In the second method the iterable need not be a literal, it can (and usually is) just a variable referring to a computed iterable. This allows for very powerful test-driven testing.

The Spock features are:

- the where clause which enforces the iteration structure over the iterable providing data.
- the `@Unroll` AST transform, which causes Spock to rewrite the code creating one test method per entry in the iterable using the method as a 'template'.

So the code as written represents 12 test methods, with the name of each of them incorporating the value of the data that the method was generated for – this is what the #i in the method name does for us. Without the `@Unroll` the test still works but it is just a single test method with iteration – not as good as the situation with the `@Unroll`.

This is surely jump up and down for joy impressive?

8. For anyone trying to undertake data-driven testing, TestNG data providers are a much nicer tool than JUnit4 parameterized tests. This is a good reason for using TestNG over JUnit4. Of course Spock is even better than TestNG, so the only choice is Spock.

there is just so much text here, especially compared to the Spock version. Java is a verbose language, and here it shows

```
package uk.org.russel.stuff

import spock.lang.Specification
import spock.lang.Unroll

import static uk.org.russel.stuff.Factorial.iterative

class Test_Factorial_Spock_Parameterized_Groovy extends Specification {
  @Unroll
  def 'iterative(#i) succeeds'() {
    expect:
    iterative(i) == r
    where:
    i | r
    0 | 1G
    1 | 1G
    7 | 5040G
    12 | 479001600G
    20 | 2432902008176640000G
    40 | 815915283247897734345611269596115894272000000000
  }
  @Unroll
  def 'iterative(#i) throws exception'() {
    when:
    iterative(i)
    then:
    thrown(IllegalArgumentException)
    where:
    i << [-1, -2, -5, -10, -20, -100]
  }
}
```

Listing 8

Getting parameterized: the TestNG Variant

So as to 'prove' the Spock way of doing things is superior in all ways, it is necessary to show an alternative. To date we have seen JUnit4 codes, but with comments that 'TestNG is better'. So now is the time for a TestNG example. Listing 9 is a test of the iterative factorial function using TestNG and its data providers.

This creates 12 distinct test methods, just as the Spock version did. However, for me, there is just so much text here, especially compared to the Spock version. Java is a verbose language, and here it shows. Coding this TestNG code in Groovy doesn't help that much because of the use of arrays and the annotations. Much as I used to love TestNG for testing, I have deserted its use for use of Spock.

Getting parameterized: the JUnit4 variant

I suggest we just do not go here. JUnit4 parameterized tests relies on use of public classes and so you have to have one test per file. In this case we would have to have two files (one for positive values, one for negative values) with most of the content the same. Let us leave this as an exercise for the reader. I can assure you that after just a short while, you will agree that the TestNG way is far superior to the JUnit4 way. The only moot point will be whether the TestNG approach is coming anywhere close to the readability, and efficacy of the Spock approach. Not so much a moot more a forgone conclusion. Spock long and prosper.

Mocking the code under test

Some people like to use mocks when unit testing, and perhaps a bit when integration tests.[9] Other claim that any use of mocks in any form of testing misses the point about what testing is and what testable software structuring is. We shall ignore this entire debate for the purposes of this article.

So why this little section? JUnit3, JUnit4, and TestNG have no notion of mock built in to the framework. Instead there is EasyMock, JMock, Mockito, an entire plethora of mocking frameworks, some of which are not at all bad. Spock though has absorbed directly, earlier work on mocks in a Groovy context: Groovy being a dynamic language, it is incredibly easy to do mocking, monkey patching, stubs, fakes, spies, etc., etc. So whilst mocking is a 'big deal' in Java, hence many sophisticated mocking frameworks using all sorts of (bizarre) reflection techniques,[10] mocking in a dynamic language is actually rather easy – but still benefits from a formalized framework, cf. unittest.mock in Python.

The point here is that dynamic languages are great languages for writing testing frameworks, whereas things can get rather complicated in static ones. Groovy is a splendid base for Spock, and Spock makes most excellent use of Groovy and its capabilities.[11]

Conclusions

I expect that you are already impressed by Spock and want to use it for all Java (and Groovy) code testing. Many people working on the JVM have had the Spock revelation, and I hope there will be more articles on Spock

9. Anyone found using mocks as part of what they claim is system or end-to-end testing, clearly need some re-education.
10. Java's reflection system exists, but is not really that good.
11. The real reason for this section is to be a bit of a tease for a future article.

in the pages of this august journal.[12] Certainly I have a few ideas for more articles, some of which will expand on the Spock theme.

Obviously the set of available testing frameworks associated with the JVM is much, much larger than I have set out here, there is ScalaTest, ScalaCheck, Specks,… the list goes on. In the main though people tend to use Scala frameworks for Scala code, Ceylon frameworks for Ceylon code, and by habit JUnit (or TestNG) for Java code. Many are though now using Spock for any Java or Groovy code. The point here is that Groovy and Java have a special relationship in that Groovy uses the Java data model directly whereas Scala, Ceylon, Kotlin do not – though these other languages are able to inter-work with Java easily (so as to access the Java Platform in its entirety), but there is an adaption layer. This is not the case with Groovy. Thus Spock, JUnit TestNG are in direct competition for testing Java and Groovy code. For me, Spock wins, hands down. Many others believe the same thing. ■

12. I think I already did the August 'joke'.

Places to look

This is a list of links (checked on 2 Jan 2016) of places for further information about Spock and the other technologies mentioned in this article:

- Java's homes: https://www.java.com, http://openjdk.java.net/
- JUnit's home: http://junit.org/
- TestNG's home: http://testng.org/
- Groovy's home: http://www.groovy-lang.org/
- Spock's home: http://spockframework.org still redirects to the now defunct Googlecode project area.

  Spock's documentation is at http://docs.spockframework.org/ which redirects to a GitHub Pages area.

  The project is active at GitHub https://github.com/spockframework.

Acknowledgements and thanks

Thanks to Frances Buontempo and the anonymous reviewers for various comments and feedback on an earlier version of this article. All the typos were fixed, but that doesn't mean there are none left! Most of the points of content led to updates to the article, but one or two I chose not to take on board. The 'ignored' topics raised lead to quite long points, that may end up as short articles in the future.

```
package uk.org.russel.stuff;

import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;
import static org.testng.Assert.assertEquals;

import java.math.BigInteger;

import static uk.org.russel.stuff.Factorial.iterative;

public final class Test_Factorial_TestNG_DataProvider_Java {
  @DataProvider
  private final Object[][] positiveData() {
    return new Object[][] {
        {0, BigInteger.valueOf(1)},
        {1, BigInteger.valueOf(1)},
        {7, BigInteger.valueOf(5040)},
        {12, BigInteger.valueOf(479001600)},
        {20, new BigInteger("2432902008176640000")},
        {40, new BigInteger("815915283247897734345611269596115894272000000000")}
    };
  }
  @DataProvider
  private final Object[][] negativeData() {
    return new Object[][]{{-1}, {-2}, {-5}, {-10}, {-20}, {-100}};
  }
  @Test(dataProvider = "positiveData")
  public void positiveArgumentShouldWork(final long n, final BigInteger expected) {
    assertEquals(iterative(n), expected);
  }
  @Test(dataProvider = "negativeData", expectedExceptions = {IllegalArgumentException.class})
  public void negativeArgumentShouldThrowException(final long n) { iterative(n); }
}
```

Listing 9