overload 119

FEBRUARY 2014 £3

Capturing lvalue References in C++11 Lambdas

We explore the subtleties of pass-by-value and pass-by-reference semantics

Integrating Catch into Visual Studio

How to integrate this popular unit test framework with Microsoft's IDE

Anatomy of a Java Decompiler

The complexities of decompilation: transforming object code back into source

Static polymorphic named parameters in C++

A description of *method chaining* — an interesting method to pass parameters into methods in a more readable fashion

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 119

February 2014

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@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 120 should be submitted by 1st March 2014 and those for Overload 121 by 1st May 2014.

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

4 Static Polymorphic Named Parameters in C++
Martin Moene demonstrates method chaining to make code readable.

7 Integrating the Catch Test Framework into Visual Studio
Malcolm Noyes uses Catch directly inside Microsoft's IDE.

11 Anatomy of a Java Decompiler
Lee Benfield and Mike Strobel transform object code back in to Java source code.

16 Optimizing Big Number Arithmetic Without SSE
Sergey Ignatchenko and Dmytro Ivanchykhin try to do arithmetic with big numbers quickly.

19 Capturing lvalue References in C++11 Lambdas
Pete Barber considers capturing references by reference and value.

Copyrights and Trade Marks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission from the copyright holder.

Random (non)sense

It's not pretty and it's not clever. Frances Buontempo considers if the cut-up method can be used to generate editorials.

It is traditional to review the last twelve months around this time of year. Having now avoided writing a proper editorial for over twelve months, I wondered if my collection of musings was worth reviewing. The thing to do was clearly try to automate the process. Having been far too busy to write an automatic editorial generator, but inspired by Charles Stross's recent blog [Stross] wherein he used Markov chains to generate text based on the King James Bible and H. P. Lovecraft leading to a strange and oddly pleasing fusion of the two styles, I found some python code to generate text [MarkovChains] and ran it over my previous excuses for editorials. A Markov chain generates a new state from the current state without looking back at history and is therefore frequently described as memoryless. Each state, for example in text processing, a word, can move to one or more other states with a pre-specified probability. The probabilities in text processing will be formed from analysis of a 'seed' document. Though this would not create a review as such, it should generate text in the spirit of the inputs. Unfortunately, this code tended to keep whole sentences or at least phrases, though it did generate some interesting 'thoughts', if one can call machine-written words thoughts.

> 'If the code is compiled, there is no documentation, or no version control.'
>
> 'I suspect I will not be changed between runs'
>
> 'get off having to write an automatic editorial generator'
>
> 'I enjoy reading sci-fi, though I do wonder why these stories still tend to insist on the idea of carrying out instructions.'
>
> 'I keep writing'

Deeply disappointed with the rehash of whole sentences, like a bad montage of television programmes at the end of the year, I then ran Stross's perl on the same input. After a couple of package installations, and filtering out all the errors, we get a variety of unhelpful or ridiculous musings, my favourites being

> 'Many then fall in love with their brains engaged.'
>
> 'Electronic wizards can be given the instructions for a four year stint.'
>
> 'C++ is provable or falsifiable.'
>
> 'The creation of the calculus gave ways to form the language, though paused for Turing.'

And near poetry

> 'If it works, it easier than an answer.
> We have sometimes taken as
> 'You have decayed away.
> Imagine that one day.
> A variety of ways of editing inputs for computers,
> so many technical books do you.'

Randomly generated machine outputs have a long and varied history. For example, Monte-Carlo simulations are frequently used to solve difficult numerical problems. This approach requires an upfront, often iterative, model wherein the next number is generated using the previous number with some degree of random perturbation, or each output generated by choosing a random input. Genetic programming, GP, attempts to automatically generate code, or even design machines such as circuit boards, by randomly piecing together shapes according to rules, be that expression trees or chips and connectors [GP]. GP needs no upfront model, but does require a pre-defined fitness function to select the better solutions to a given problem. It starts with a generation of randomly created solutions and cuts them up, referred to as crossover, sometimes randomly mutating parts, to reform other candidate solutions from the pieces, supposedly thereby mimicking evolution.

Unfortunately, it is difficult to decide a model for generating an editorial in advance, or give a precise fitness function for acceptable editorial attempts. This does not preclude the possibility of using randomness to create text, poems, or indeed other forms of art. Having (nearly) stuck to my book buying ban this year, I persuaded my sister to buy me *Assimilate: A Critical History of Industrial Music* [Assimilate] (not recommended for the faint-hearted). It delves into the artistic background and precedents for various noisy 'metal machine music' [MMM] bands. The original Lou Reed album is often described as, 'Electric guitars feeding back to create a complex multi-layered sound collage' [BBC], though the phrase has become almost legendary appearing in songs and titles by Die Krupps, Sabaton and others. *Assimilate* suggests many approaches to creating industrial music trace back to William S Burrough's 'Cut-up' method: "The cutup is a mechanical method of juxtaposition in which Burroughs literally cuts up passages of prose by himself and other writers and then pastes them back together at random" [Cut-up]. This in turn can be traced back to the Dada movement.

Frequently, legacy code bases appear to have been formed using a similar cut (and paste) approach. Snippets of functions proliferate through the entire code base peppered with a variety of mutations on the way. Other functions fail to follow the artistic coding style, if there is one, for example breaking brace placement or white space conventions, suggesting a manual attempt to slam randomly selected code samples taken from elsewhere into the mix to solve a problem (possibly accidentally creating another). People usually leave typos intact when doing this, making it easy to trace the provenance of the code. If only they'd copy over any unit tests when taking such an approach. I believe it is traditional to ask when

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 has been a programmer for over 12 years professionally, 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.

confronted by such modernist techniques and 'installations', "But is it art?"[1]

The final result can appear like a discordant, jarring mess speckled with repeated leitmotifs. This mirrors the disorienting effect of some of the more extreme, experimental industrial noise artist's outpourings. Where the music is a bid to either block out the world or to escape the claimed 'viral impact' of convention by shaking people out of their norms, the code is usually just an endeavour to implement some new features, though it may have a similar slightly nauseating impact on its audience.

Applying the cut-up coding methodology is impossible without an input stream of code to copy and paste. Nowadays people tend to use the internet as the source of all source, so in the spirit of Dadaist impishness, it can be lots of fun to unplug network cables at random. Since the Dada movement has been described as "flout[ing] conventional aesthetic and cultural values by producing works marked by nonsense, travesty, and incongruity" [Dada], there would be delicious irony in apply Dadaist techniques in order to enforce aesthetic and cultural values in a code base, thereby keeping the flame [KoF] and stopping "nonsense, travesty and incongruity". I suggest a New Year's resolution to attempt to write some code from time to time with your network cable unplugged.

If one already has a code base, that can be used directly as an input for genetic programming. Or, in lieu of a fully formed GP application, other types of randomness can still be applied fruitfully. Indeed, a partial step towards genetic programming becoming common-place is the recent interest in mutation testing. This takes and mutates existing source code, then running it against a suite of tests, which are functioning like the fitness function in GP. The mutations may swap binary logical or arithmetic operators, such as `&&` with `||` or `+` with `-`, delete statements, or swap variables in the same scope. Various other mutations are possible. The only difference to GP is the lack of cross-over and the original code is human generated, rather than randomly generated by a machine. For example, in the Java based PITest,

> Faults (or mutations) are automatically seeded into your code, then your tests are run. If your tests fail then the mutation is killed, if your tests pass then the mutation lived. The quality of your tests can be gauged from the percentage of mutations killed. [PITest]

This is therefore a way of testing the tests. Any living mutations can suggest further tests that need adding, or requirements that need clarifying, or if you are very lucky code that can be deleted. This supposes you have some tests. Applications do exist to write tests for you, for example Microsoft's Pex and Moles [Pex] though I suggest you do write some of your own tests first, preferably before writing any code let alone before using Pex. I assure you the potential edge cases you may have missed if you do not are legion. Being presented with millions of tests which fail from a few thousand lines of code is overwhelming. Nonetheless either randomly mutating code or randomly generating tests can be very informative.

It is difficult to draw concrete conclusions from these thoughts, so I ran Stross's Perl[2] code [op cit] over the above. It produced two revelations:

> Deeply disappointed with repeated leitmotifs
>
> The claimed 'viral impact' of convention by shaking people out of mutations'

It seems my automatic editorial generator is a long way off. However, parallels between the ways in which some code bases develop, various art movements in the last hundred years and differing aesthetic viewpoints is interesting. The creative process is fascinating, whether applied to music, art or computer programming. Randomly shaking things up can give results. The results may not always be pleasing, but can be thought provoking and might just work. Whether everyone is in agreement about the final result is another matter. Some will scream in horror, "Make it stop!" while others may be delighted with the outcome. We are, after all, frequently told "Beauty is in the eye of the beholder."

1. Attributed to Rudyard Kipling as either "It's pretty, but is it art?" or "It's clever, but is it art?" Frequently it's neither pretty nor clever.

References

[Assimilate] *Assimilate: A Critical History of Industrial Music* S. Alexander Reed, Oxford University Press 2013

[BBC] http://www.bbc.co.uk/music/reviews/wzwx

[Cut-up] http://www.languageisavirus.com/articles/articles.php?subaction=showcomments&id=1099111044&archive=&start_from=&ucat=#.UsK8z7QobYs

[Dada] http://www.thefreedictionary.com/dadaist

[GP] http://www.genetic-programming.com/

[KoF] http://c2.com/cgi/wiki?ArchitectAsKeeperOfTheFlame

[MarkovChains] http://www.decontextualize.com/teaching/rwet/n-grams-and-markov-chains/

[MMM] http://en.wikipedia.org/wiki/Metal_Machine_Music

[PITest] http://pitest.org/

[Pex] http://research.microsoft.com/en-us/projects/Pex/

[Stross] http://www.antipope.org/charlie/blog-static/2013/12/lovebiblepl.html

2. Possibly spelled PERL or `rm -f /usr/bin/perl`

Static Polymorphic Named Parameters in C++

Adding parameters to an object can be messy. Martin Moene demonstrates how method chaining can make code more readable.

For a new kind of measurement in our application for scanning probe microscopy [Wikipedia-a], I need to construct a curve that consists of several kinds of segments. The curve can for example describe the movement of the tip perpendicular to the surface of the material investigated and what data shall be acquired.

Behaviour of segment types varies. One segment may describe how the surface is approached, another how to move away from the surface and yet another describes a dwell time. Such a curve is part of force-distance spectroscopy [Wikipedia-b]. Figure 1 below shows what the researchers would like to do.

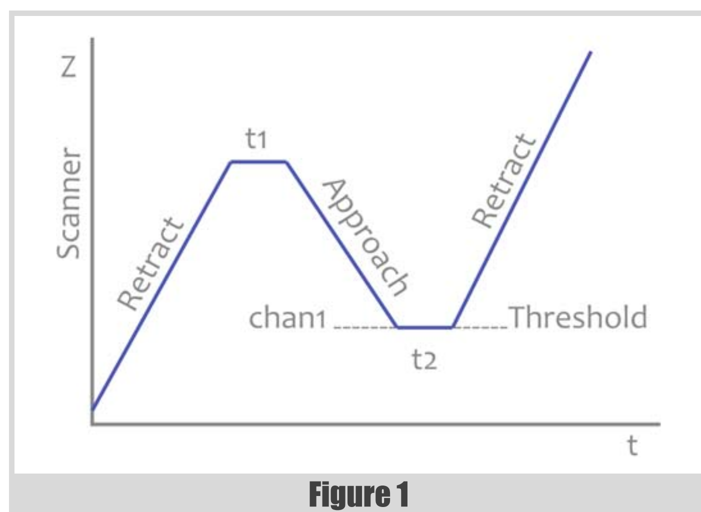


Figure 1

To a large extent the structure of a curve is fixed and this structure can be created at compile-time. Some variation is required at run-time, which can be arranged for via parameters. The simplified code in Listing 1 illustrates this.

The curve built describes that: Nsweep times, scanner `Z` retracts the tip from the material surface for 123 nm (unless skipped), then approaches the surface until the measured value of `chan1` reaches the threshold value of 2.7 V, and finishes with retracting 123 nm again (t1 and t2 are omitted). Note that the threshold condition depends on other information than the scan distance. It can have any unit that makes sense in the experiment.

In the real implementation, there are many more parameters. To keep it simple, several things such as data acquisition are omitted here. Another simplification is to use struct without access specifiers for all classes in the code.

Compiling and running the simplified program gives:

Martin Moene has a background in electronics engineering and has been programming professionally since 1983, mostly in C++. Much programming revolves around instrument control and image processing and he enjoys seeing elegance in code. Martin Moene can be contacted at m.j.moene@eld.physics.LeidenUniv.nl.

```
#include "curve.hpp"

int main()
{
  // run-time configurable:
  const int  Nsweep = 1;
  const bool skipR1 = false;
        auto scanner   = create_scanner   ( "Z" );
  const auto distance  =
     create_condition( "123 nm" );
  const auto threshold =
     create_condition( "chan1", "<=", "2.7 V" );

  Curve curve;

  curve.times( Nsweep   )
    .scans( scanner )
    .add   ( Retract  ().stop_on( distance   ) )
       .unless( skipR1 )
    .add   ( Approach().stop_on( threshold ) )
    .add   ( Retract  ().stop_on( distance   ) )
  ;
  std::cout << "curve.sweep(): "; curve.sweep();
}
```

Listing 1

```
prompt>g++ -Wall -Wextra -Weffc++ -std=c++11 \
-o curve.exe curve.cpp && curve
curve.sweep(): RZL AZ<= RZL
```

The output `RZL AZ<= RZL` indicates the type of segment, scanner and condition used for each segment: `R` for Retract, `A` for Approach, `Z` for ZAxisScanner, `<=` for LessEqualCondition and `L` for LengthCondition.

Fluent interface

The construct sketched above evolved from the following (simplified) C++98 code (Visual C++ 6).

```
return CurveDefinition().
  sweep( sweepCount ).
  add( CurveSegmentPtr( new SweepCurveSegment
     ( scanner, condFalse, ... ) ) ).
  add( CurveSegmentPtr( new SweepCurveSegment
     ( scanner, condition, ... ) ) ).
  add( CurveSegmentPtr( new SweepCurveSegment
     ( scanner, condFalse, ... ) ) );
```

Although the sketched curve may be adequate for many kinds of experiments, it is a simplification of a more general approach. Experience with an initial version of the code led the researchers to express several additional wishes. For example to be able to conditionally include a segment, to only perform it once, or to perform a collection of segments multiple times.

the reason to compose the curve in this way is to benefit from a clear and flexible notation

As you see, the new code expands on the use of method chaining [Wikipedia-d], a key element of a fluent interface [Wikipedia-e]. Method chaining is also known as the named parameter idiom. In addition to method chaining, other variations are imaginable, such as function-like modifiers. For example to include a segment in the first sweep only with `once( segment )` or to perform a collection of segments (a sub-curve or section) a number of times via `times( N, section )`.

The new code also moves the allocation of segments out of the fluent interface. This makes the code much more readable. It also leads to the main subject of this article: static polymorphic named parameters.

Thus, *the* reason to compose the curve in this way is to benefit from a clear and flexible notation. As an internal domain-specific language [Fowler08] it also helps researchers to recognise the curve they sketched in the code.

Static polymorphism

Now, let's examine how a curve is constructed. Curve's method `add()` creates dynamic segment objects from the non-dynamic temporary 'exemplars'. To create a dynamic copy of the segment of the original type, `add()` is templated.

```
template< typename T >
Curve & Curve::add( T const & segment )
{
  segments.emplace_back(
    std::make_unique<T>( segment ) );
  return *this;
}
```

Note: `std::make_unique<T>()` is a C++14 feature [make_unique].

Looking at above code, it becomes clear that a call like `Approach().stop_on(...)` must itself return an object of type `Approach` to add the right type of segment to the curve.

Here is the crux of this article. Do all types such as `Approach` require their own method `stop_on()` to return the appropriate type? Fortunately that's not the case, thanks to the curiously recurring template pattern or CRTP. See [Wikipedia-e: Subclasses] and [Wikipedia-f] respectively. (See Listing 2.)

With this construct `Approach` can inherit `stop_on()` that returns the desired `Approach &` instead of `SegmentParameter &`. The `crtp_cast` combined with a macro enables us to write `return self` to return the current object *with the right type* where we would otherwise write `return *this` [Bendersky11]. The shortest of four `crtp_cast` const-volatile variations is:

```
template<class D, class B>
D & crtp_cast(B & p)
{ return static_cast<D &>( p ); }
```

For an interesting discussion about encapsulation and the CRTP, see Better Encapsulation for the Curiously Recurring Template Pattern by Alexander Nasonov [Nasonov05].

```
template <typename Derived>
struct SegmentParameter : SegmentCommon
{
#define self crtp_cast<Derived>(*this)

    Derived & stop_on( ConditionPtr cond )
    {
        condition( cond );
        return self;
    }

#undef self
};

struct Approach : SegmentParameter<Approach>
{
   // inherited:
   // Approach & stop_on( ConditionPtr s );
};
```

Listing 2

Build to use

In the end we've built a curve that contains a collection of smart-pointered segments that originate in interface `Segment`. At the same time the curve is a segment itself, so that it can act as a sub-curve or section of another curve. See the following derivation chains.

```
Curve → SegmentParameter<Curve>
  → SegmentCommon →  Segment
Approach → SegmentParameter<Approach>
  → SegmentCommon → Segment
...
```

Thus, whereas construction of the curve builds on automatic 'exemplar' objects and static polymorphism via the CRTP compile-time technique, using the curve occurs via classical dynamic polymorphism with `Segment` as the interface.

Letting go of the garbage

One little thing worries me: the code for the sequence `curve.add(...).unless(...)` is both elegant and inelegant at the same time. It is simple, but then it lets you create a segment to only throw it away immediately via `unless()`.

```
Curve & unless( bool skip )
{
  if ( skip )
    segments.pop_back();
  return *this;
}
```

One can argue that recycling is good and more garbage means more recycling

One can argue that recycling is good and more garbage means more recycling, but that isn't entirely in line with Bjarne Stroustrup's idea [Kalev13]:

> So, I say that C++ is my favorite GC language because it generates so little garbage.

As a reviewer pointed out, one may circumvent the awkward situation by prefixing the condition, or by including it in a conditional add function like so:

```
curve.enable_if_not( skipR1 )
  .add( Retract ().stop_on( distance ) );
curve.add_if_set( performR1,
  Retract ().stop_on( distance ) );
```

However, to ease reading of consecutive lines, I'd prefer to keep the left part of the lines similar, as illustrated here.

```
curve.add_if( Retract ().stop_on( distance  ),
              performR1 )
     .add   ( Approach().stop_on( threshold )
);
```

We're leaving the realms of fluid interfaces and named parameters though.

Summary

In this article a fluent interface is applied to a simplified setting for scanning probe spectroscopy. The resulting code shows a close relationship to the graphic representation provided by the researchers. The main point of the article is to show how one can obtain the static inheritance required for named parameters in this setting via the curiously recurring template pattern. ■

Acknowledgements

I'd like to thank the *Overload* team for reviewing the article and Jonathan Wakely for clarifying several aspects of smartpointers. Their remarks and suggestions were key in improving the article.

Notes and references

Code for this article and code for a larger example with modifiers **once** and **times** is available on [GitHub].

[Arena12] Use CRTP for polymorphic chaining. Marco Arena. 29 April 2012. http://marcoarena.wordpress.com/2012/04/29/use-crtp-for-polymorphic-chaining/ (Presents a slightly different application of the CRTP.)

[Bendersky11] The Curiously Recurring Template Pattern in C++. Eli Bendersky. 17 May 2011. http://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c/ (Mentions **crtp_cast** in a comment.)

[Fowler05] Fluent Interface. Martin Fowler. 20 December 2005. http://www.martinfowler.com/bliki/FluentInterface.html

[Fowler08] Domain-Specific Language. Martin Fowler. 15 May 2008. http://martinfowler.com/bliki/DomainSpecificLanguage.html

[GitHub] Code for Static polymorphic named parameters in C++. Martin Moene. 31 December 2013. https://github.com/martinmoene/martin-moene.blogspot.com/tree/master/Static polymorphic named parameters in C++

[K-ballo13] Episode Eight: The Curious Case of the Recurring Template Pattern. *Tales of C++ K-ballo*. 2 December 2013. http://talesofcpp.fusionfenix.com/post-12/episode-eight-the-curious-case-of-the-recurring-template-pattern

[Kalev13] An Interview with Bjarne Stroustrup. Danny Kalev and Bjarne Stroustrup. May 15, 2013. http://www.informit.com/articles/article.aspx?p=2080042

[make_unique] make_unique. *CppReference*. http://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique Here, function **make_unique<>()** is equivalent to **std::unique_ptr<T>(new T(std::forward<Args>(args)...))**. See also Herb Sutter's GotW #89 Solution: Smart Pointers (http://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/) and GotW #102: Exception-Safe Function Calls (http://herbsutter.com/gotw/_102/).

[Nasonov05] Better encapsulation for the curiously recurring template pattern. Alexander Nasonov. **Overload**, 70:11-13, December 2005 (http://accu.org/index.php/journals/296).

[Wikipedia-a] Scanning Probe Microscopy. *Wikipedia*. http://en.wikipedia.org/wiki/Scanning_probe_microscopy Accessed 21 December 2013.

[Wikipedia-b] Force-Distance spectroscopy . *Wikipedia*. http://en.wikipedia.org/wiki/Atomic_force_microscopy#Force_spectroscopy Accessed 19 December 2013. See also [Wikipedia-c].

[Wikipedia-c] Scanning tunneling spectroscopy. *Wikipedia*. http://en.wikipedia.org/wiki/Scanning_tunneling_spectroscopy Accessed 19 December 2013.

[Wikipedia-d] Method chaining or named parameter idiom. *Wikipedia*. http://en.wikipedia.org/wiki/Method_chaining Accessed 16 December 2013. See also [Wikipedia-e].

[Wikipedia-e] Fluent interface. *Wikipedia*. http://en.wikipedia.org/wiki/Fluent_interface Accessed 21 December 2013. See also [Fowler05]

[Wikipedia-f] Curiously recurring template pattern (CRTP). *Wikipedia*. http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern Accessed 17 December 2013. See also [K-ballo13], [Bendersky11], [Arena12].

Integrating the Catch Test Framework into Visual Studio

Visual Studio's Test Explorer allows native C++ tests to be run under a managed wrapper. Malcolm Noyes takes it a step further using Catch to drive the tests.

Recently I adapted Phil Nash's Catch C++ (and Objective C) testing framework [Nash] to integrate with Visual Studio (VS). I've made a fork of Catch available on Github [Catch], together with some documentation that explains how to use it in that environment [VS]. I thought perhaps that for ACCU it would be more interesting if I wrote up some details about why it does what it does.

For those who are unfamiliar with C++ testing in Visual Studio...

The five minute guide to testing in Visual Studio

First, I should define some terms that I'll be using. A 'Managed' C++ test is one that runs native C++ code (the code we want tested) under a managed C++ wrapper (that creates the test environment). Until VS2012 this was the only kind of C++ unit test that you could write that integrated with the Visual Studio IDE. With VS2012, Microsoft added 'Native' C++ tests. These are tests that use a native C++ wrapper to create tests but that can still be run from the Visual Studio IDE.

Figure 1 shows an example of a 'Managed' test in VS2012. The IDE has the Test Explorer to the left showing that the test has failed. Clicking the highlight at the top of the stack trace (bottom left) opens the code and positions the cursor at the failing line (I've manually highlighted the line to make this clearer...).

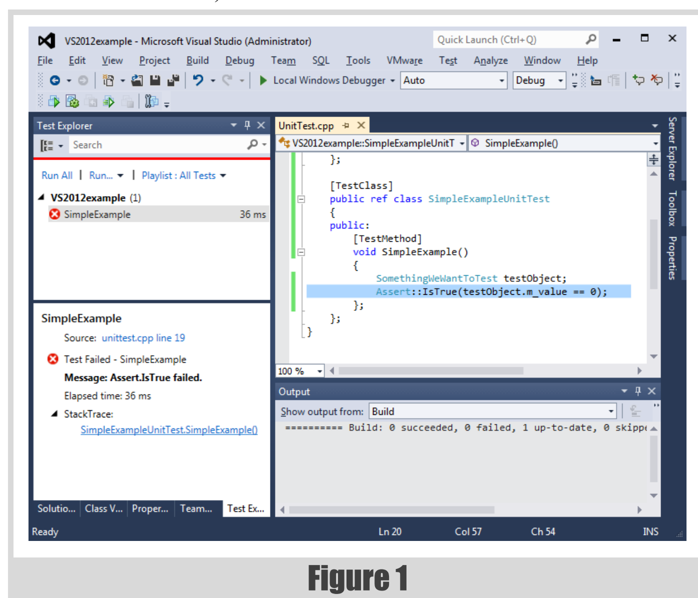


Figure 1

This is what I wanted to replicate with Catch...so for those who are unfamiliar with Catch....

The five minute guide to Catch

Catch is a C++ testing framework that is simple to get running (header only, no dependencies) and in the case of failure (or optionally for success) can also provide both the original expression and the values that caused failure. The current version is designed to run from the command line.

```
// example_tests.cpp
#include "catch.hpp"

struct SomethingWeWantToTest {
  SomethingWeWantToTest() : m_value(1) {}
  int m_value;
};

TEST_CASE("Simple example") {
  SomethingWeWantToTest testObject;

  SECTION("First section, fails") {
    REQUIRE(testObject.m_value == 0);
  }
  SECTION("Second section, works") {
    REQUIRE(testObject.m_value == 1);
  }
}
```

Listing 1

To make the command line work, Catch needs a **`main()`** function; my personal convention is to create a main.cpp with this content:

```
// main.cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

Then I create a file for my tests (this file can be shared with Visual Studio) and write a test (see Listing 1).

The 'test case' is defined with free form text for the name, then creates an instance of the object that we want tested. When the tests are run, Catch will loop through the **`TEST_CASE`** for as many **`SECTION`**s as are defined, so in this example the **`TEST_CASE`** gets run twice. This creates a new, initialised **`testObject`** each time the **`TEST_CASE`** is run; for this reason many Catch tests require no **`setup()`** or **`teardown()`** methods.

The first **`SECTION`** will clearly fail, but Catch carries on and runs the **`TEST_CASE`** again to run the second **`SECTION`**, which works. The output from a test run with default arguments is like Listing 2.

For the failure, the program outputs both the original expression and the values that caused the failure. The final line confirms that the assertion in each **`SECTION`** was executed, with 1 failure. Catch can also be run to show the output from successful **`REQUIRE`**ments, along with many other options.

Malcolm Noyes has worked as a software developer/author for several years; just how many can be deduced from the information that he started programming C++ using a Zortech compiler. He wrote several string classes before discovering the STL and several thread classes before multi-threading got standardised. He has never written a Unit Test framework but probably would have done if Phil Nash hadn't got there first!

I had written some macros that enabled me to share source code between Catch and MSTest but this didn't integrate very well with the IDE

```
test.exe is a Catch v1.0 b23 host application.
Run with -? for options

-------------------------------------------------
Simple example
  First section, fails
-------------------------------------------------
c:/Projects/examples/example_tests.cpp:8
.................................................

c:/Projects/examples/example_tests.cpp:12: FAILED:
  REQUIRE( testObject.m_value == 0 )
with expansion:
  1 == 0

=================================================
1 test case - failed (2 assertions - 1 failed)
```

Listing 2

Goals using Catch in Visual Studio

My initial goal was purely selfish; I currently work in an environment where VS is used to implement tests using Microsoft's test framework. The environment has a couple of serious usability problems [MSTest] and this makes testing a somewhat painful experience.

My involvement started from Visual Studio 2010 and later moved on to VS2012. From a fairly early stage I had written some macros that enabled me to share source code between Catch and MSTest but this didn't integrate very well with the IDE, then I discovered by accident that VS2012 had some 'Native' C++ unit test support and I realised that it should be possible to hook into this directly from Catch. This started a train of thought that made me wonder if I could do the same thing for Managed tests too.

Initially I wanted:

1. To be able to write a test with Catch macros and share source code between command line Catch and VS.
2. If an assertion failed in the IDE, the test should stop and the IDE should allow me to jump to the location of the problem, just like it does in MSTest.

This last requirement is slightly different from running regular Catch from the command line; normally we expect that Catch will do its best to run as many of the tests as possible, then report all of the problems at the end of the run. In the IDE, I wanted it to stop and that meant that I had to tinker with some of Catch's internals...

First implementation

So I spent a couple of days doing an experiment and ended up with something that seemed to meet these goals. To make it work, I had to do three things:

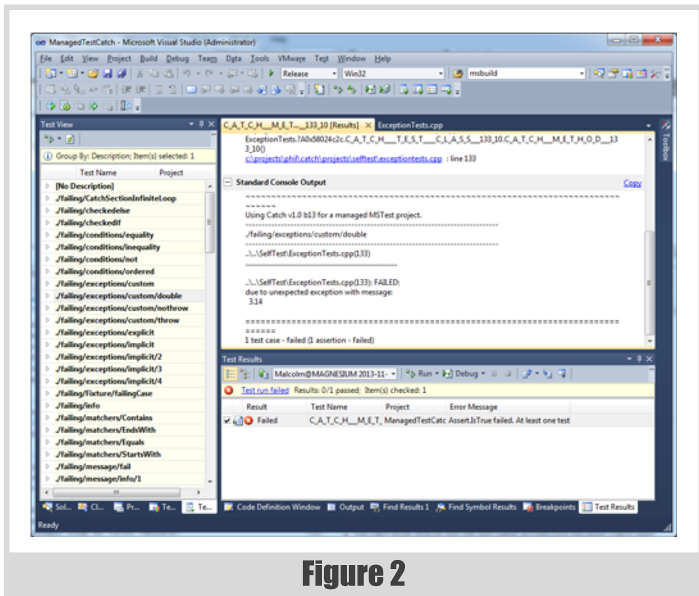


Figure 2

1. Redefine the **TEST_CASE** macro.
2. Rewrite the reporter so that it reported at the end of the test.
3. Hook up the assertion to the relevant MSTest mechanism.

The result looks like Figure 2 (VS2010).

Changes to the TEST_CASE macro

The **TEST_CASE** macro maps to an underlying **INTERNAL_CATCH_TESTCASE** macro that needed a completely new definition, along with the equivalent macros for testing class methods that are very similar.

My first version of these macros implemented an instance of a test class (either a 'Managed' test class or a 'Native' test class), provided a semi-unique name for a function that would get invoked and then called it (more about the 'semi'-uniqueness later...). As in regular Catch, the definition of the invoked **TEST_CASE** function follows the macro and is written by the user. The Catch 'test name' is passed as a property attribute to the test class so that it can be displayed in the IDE. In the first version, the Catch 'description' field was discarded.

```
// TEST_CASE maps to INTERNAL_CATCH_TESTCASE
// First param is test name
// Second param is 'description',
// often used for tags
TEST_CASE( "./failing/exceptions/double",
 "[.][failing]"  )
{
   //...
}
```

Since Catch is header only, I also had to adapt the code to allow for functions that would be **#include**d multiple times; Regular Catch

if there is a duplicate name, VS silently ignores one of them and only runs one of the tests

'knows' which module contains **main()** so only includes certain headers in that module. Using an MSTest project in Visual Studio doesn't require a **main()** function, so catch.hpp needs to pull in everything in each module. Consequently, some functions needed to get inlined and a couple of static members needed to get replaced by templates so that the compiler/linker had to work out how to keep just one instance from all modules.

Changes to the reporter

The first reporter looked very similar to the Catch 'console' reporter. The main change was to collect together all the information that needed to be reported when the test completed so that it could be sent to the output windows in VS....

...except that Phil had optimised a couple of functions that returned strings using a static **std::string**. It turns out that Managed C++ doesn't like this much ('This function must be called in the default domain' in atexit, presumably as a result of trying to release the memory from the **std::string**). So (for now) those functions had to return a new string each time. Ho hum...

Assertions

Catch has a number of macros that check for failures (**REQUIRE**, **CHECK**, **REQUIRE_FALSE**, etc) but underlying all of this is an **INTERNAL_CATCH_ACCEPT_EXPR** macro that helpfully throws an exception if the test needs to abort. All I had to do to make this work was call the MSTest assertion with the relevant details of the failure instead of throwing. That would give me the context that I wanted so that the IDE could point me to the problem by clicking the link.

Some feature creep

Catch already has an extensive internal self test suite that is slightly complicated because, being a test framework, needs to check failures as well as tests that pass. I had decided that the best way to check that my code worked would be to build a project that used all of the internal test suite and one of the first modules I encountered was the VariadicMacrosTests file.

I had already started to prepare a blog post about what I had done and as part of that I created a new VS test project from scratch, using the VS wizard. However I noticed that the wizard generated a project that used Unicode by default. I wanted to be able to use multi-byte strings (MBCS) as well. 'Simple', you may think and indeed it was until I encountered the variadic macros in Catch. The problem with this is that some of the strings needed to be passed as wide strings (e.g. to test class attributes) and some didn't (e.g. to be used internally by Catch). When it isn't known how many parameters have been passed to the macro, it can become tricky to know how to convert a possibly non-existent value! Much of the complexity in my implementation of the **TEST_CASE** macros is there to deal with this problem.

I also discovered that Native C++ tests didn't want to play nicely with anonymous namespaces. I tried several possible ways to fool the compiler into allowing a unique class name to be used but in the end I had to accept that each test needed to go into a namespace uniquely named for each file. If I didn't do this, there was a risk that the semi-unique name generated by the **TEST_CASE** macro would cause a name clash between modules.

Sadly if there is a duplicate name, VS silently ignores one of them and only runs one of the tests, so I had to manually check for a name clash and generate an error if it happened. The workaround for both these problems is pretty simple; since neither Catch or VS cares what namespace the tests are in, **TEST_CASE**s in a module should go into a namespace named after that module, e.g.:

```
// module1.cpp
namespace module1 {
    TEST_CASE("blah") {
        //...
    }
}
```

Remarkably, most other things just worked.... but I had some trouble with Catch's ability to register and translate unknown exceptions...

More feature creep

Although I suspected that the feature wasn't used much, I was sure that it should be possible to implement some code that would allow me to translate unknown exceptions. I realised that to make it work I would have to fix up Catch's static registration. The existing macros worked like this; first define a translation:

```
CATCH_TRANSLATE_EXCEPTION( double& ex )
{
  return Catch::toString( ex );
}
```

then when a test throws an unexpected exception, it should be sent to the output, e.g.:

```
TEST_CASE( "Unexpected exceptions can be
translated", "[.][failing]"  )
{
  if( Catch::isTrue( true ) )
    throw double( 3.14 );
}
```

will send this to the output:

```
...
c:\projects\catch\projects\selftest\exception
tests.cpp(130): FAILED:
due to unexpected exception with message:
  3.14
```

In common with many other test frameworks, Catch has a global registration object that it uses to register tests, and it also uses this to register reporters and exception translators, so initially I just implemented the exception translators to use a static templated object.

Once I'd done that I could run each individual test from the self test suite in the IDE and manually check that the output matched the output from Catch.

A fortunate co-incidence

I had been thinking about one of the other tests in the test suite that I hadn't managed to implement. The code collected all the registered tests, worked out whether they were expected to 'pass' or 'fail' and then ran them all in two batches, one for passes and one for failures. Around this time, Phil had written a Python script that verified the expected output from running all tests, so he removed this code. I had a feeling this code might be useful, and so it prompted me to think about how I could perhaps use a similar technique to run all the tests automatically in VS. My idea was to run all the tests in a 'batch', then use a similar Python script to generate compatible output from the VS output, instead of having to manually check every time.

However, in that version I didn't register any tests; my initial implementation of **TEST_CASE** just created a test class and ran the method inline, so didn't need it. After a little tinkering with a few angle brackets, I discovered that I could indeed implement the whole of Catch's registration mechanism in VS. Then I started to wonder what else I could do with it...

A new goal!

I started with a new macro that didn't register a catch test case but instead asked the Catch registration object to run all the tests. That worked but I knew that Phil was able to use Catch's 'tag' filtering to selectively run different groups of tests and I wanted to be able to do the same thing. I also realised that if I could somehow call this 'batch test' using the VS command line tools then I would be able to easily integrate tests written for Catch into other Continuous Integration environments, such as TeamCity, TFS or Jenkins.

During my first encounter with Native C++ tests I had discovered that VS2012 Native tests could not use MSTest.exe to run them from the command line. Instead it seemed that MSTest.exe had been deprecated in favour of vstest.console.exe that had the necessary plumbing to understand binaries built for native tests and run them. A brief look at the command line parameters for both tools suggested the options for passing filters into tests was going to be limited to a single textual parameter ('Category' for MSTest and, somewhat bizarrely, 'Owner' for Native vstest.console.exe).

I did explore the possibility of feeding parameters into the tests as a database but that seemed overly convoluted and it wasn't clear if it would work in Native C++ tests (I don't think it does...) so I ended up with a macro whereby I could specify an identifier that would be recognised by the VS tools and that I could 'map' to Catch 'tags' to provide filtering using the existing Catch code, the snappily named **CATCH_MAP_CATEGORY_TO_TAG**:

```
CATCH_MAP_CATEGORY_TO_TAG(all, "~[vs]");
```

This runs all the registered tests except those tagged '[vs]', which corresponds to the default run of Catch on the command line. This macro also changes the default behaviour of Catch so that instead of stopping immediately (as we need for the IDE) it runs as many tests as possible. Sadly, this change in behaviour also exposed some shortcomings in my implementation of the reporter and capture mechanisms; in some circumstances expected output would be lost. This required a bit more rework of Catch's internals, in particular I found that I needed to push the current test state onto the stack so that the reporter could use that information when a failing test was unwound.

Then I developed some new Python scripts that parsed and verified the Catch output and compared that against the output from MSTest.exe/ vstest.console.exe (output from these tools can be directed to a .trx file). Aside from some minor presentation differences, this worked well and was good enough to validate that the VS code is doing the right thing, at least for 'all' tests.

The final section

There were still two things that I couldn't validate though; the Catch self test validation script runs a set of tests that shows output from successful tests, and another that aborts after 4 failures. All these things can be easily changed using different parameters from the Catch command line. Could I replicate this somehow? What I wanted was to be able to define test parameters before I used **CATCH_MAP_CATEGORY_TO_TAG**. Such changes should apply to the current batch test run only; subsequent test runs should revert to defaults. A few additional macros and some additional configuration classes allowed me to do this too:

```
CATCH_CONFIG_SHOW_SUCCESS(true)
CATCH_CONFIG_WARN_MISSING_ASSERTIONS(true)
CATCH_MAP_CATEGORY_TO_TAG(allSucceeding, "~[vs]");
```

This produced all the correct output, but in the wrong order. The order that the tests are run depends on their order of registration with the global registrar, which of course depends on the order the compiler/linker decides to implement static objects. Phil normally uses OSX to develop Catch, and this does things in a different order from VS. The solution is to sort the output by test name in the validation scripts before it can be compared.

The same problem afflicts the test that aborts after 4 failures, but the effect is slightly different. The code for OSX was presumably written using XCode/Clang and the tests that code decides to execute before it gets to 4 failures was different to VS and I got a completely different set of failures! So finally I had to add two more macros; one that 'registers' a test to be run in a specific order and one that runs the ordered list of tests (Listing 3).

Wrap up

I now have a very flexible test environment that I can use to share Catch source code between Visual Studio and command line Catch. If I want to avoid the torture of the Test Explorer, I can run Catch from the command line by simply adding a `main.cpp` that specifies **CATCH_CONFIG_MAIN**. For those times when I need to resort to the debugger, I can easily run individual tests in the IDE. As a bonus, I can also specify a 'batch run' that uses the built in VS command line tools, which means that integration with TeamCity (or other CI environments) should be easy. I think my goals have been met; if you are suffering from similar frustrations, please give the fork a try and let me know what works, and what doesn't.

Finally, I've been discussing with Phil the possibility that the code for my fork could be merged back into the mainline; my understanding is that he is keen to do this, although as far as I know he hasn't had time to take a good look at what I've done to his code yet! So I hope that this will happen, or perhaps will have happened by the time you read this... ■

References

[Catch] My fork of Catch: https://github.com/colonelsammy/Catch

[Nash] Phil Nash's Catch framework: https://github.com/philsquared/Catch

[VS] Documentation for VS integration: https://github.com/colonelsammy/Catch/blob/master/docs/vs/vs-index.md

[MSTest] http://www.graoil.co.uk/blog/2013/10/28/replacing-mstest-with-phil-nashs-catch-framework-for-managed-tests/

```
CATCH_INTERNAL_CONFIG_ADD_TEST("Some simple comparisons between doubles")
CATCH_INTERNAL_CONFIG_ADD_TEST("Approximate comparisons with different epsilons")
CATCH_INTERNAL_CONFIG_ADD_TEST("Approximate comparisons with floats")
...
INTERNAL_CATCH_MAP_CATEGORY_TO_LIST(allSucceedingAborting);
```

Listing 3

Anatomy of a Java Decompiler

Java byte code can be reverted back into source code. Lee Benfield and Mike Strobel show how.

A decompiler, simply put, attempts to reverse the transformation of source code to object code. But there are many interesting complexities – Java source code is structured; bytecode certainly isn't. Moreover, the transformation isn't one-to-one: two different Java programs may yield identical bytecode. We need to apply heuristics in order to get a reasonable approximation of the original source.

(A tiny) bytecode refresher

In order to understand how a decompiler works, it's necessary to understand the basics of byte code. If you're already familiar with byte code, feel free to skip ahead to the next section.

The JVM is a *stack-based machine* (as opposed to a register-based machine), meaning instructions operate on an evaluation stack. Operands may be popped off the stack, various operations performed, and the results pushed back onto the stack for further evaluation. Consider the following method:

```
public static int plus(int a, int b) {
  int c = a + b;
  return c;
}
```

Note: All byte code shown in this article is output from `javap`, e.g., `javap -c -p MyClass`. (Comments added for clarity.)

A method's local variables (including arguments to the method) are stored in what the JVM refers to as the *local variable array*. We'll refer to a value (or reference) stored in location **#x** in the local variable array as 'slot #x' for brevity (see JVM Specification §2.6.2 [JVMa]).

For `instance` methods, the value in slot #0 is always the `this` pointer. Then come the method arguments, from left to right, followed by any local variables declared within the method. In the example above, the method is static, so there is no `this` pointer; slot #0 holds parameter `x`, and slot #1 holds parameter `y`. The local variable `sum` resides in slot #2.

```
public static int plus(int, int);
 Code:
  stack=2, locals=3, arguments=2
   0: iload_0
      // load 'a' from slot 0, push onto stack
   1: iload_1
      // load 'b' from slot 1, push onto stack
   2: iadd
      // pop 2 integers, add them together,
      // and push the result
   3: istore_2
      // pop the result, store as 'c' in slot 2
   4: iload_2
      // load 'c' from slot 2, push onto stack
   5: ireturn
     // return the integer at the top of the stack
```

Listing 1

It's interesting to note that each method has a max stack size and max local variable storage, both of which are determined at compile time.

One thing that's immediately obvious from here, which you might not initially expect, is that the compiler made no attempt to optimise the code. In fact, `javac` almost never emits optimised bytecode. This has multiple benefits, including the ability to set breakpoints at all executable source lines: if we were to eliminate the redundant load/store operations, we'd lose that capability. Thus, most of the heavy lifting is performed at runtime by a just-in-time (JIT) compiler.

Decompiling

So, how can you take unstructured, stack-based byte code and translate it back into structured Java code? One of the first steps is usually to get rid of the operand stack, which we do by mapping stack values to variables and inserting the appropriate load and store operations.

A 'stack variable' is only assigned once, and consumed once. You might note that this will lead to a *lot* of redundant variables – more on this later! The decompiler may also reduce the byte code into an even simpler instruction set, but we won't consider that here.

We'll use the notation `s0` (etc.) to represent *stack variables*, and `v0` to represent true local variables referenced in the original byte code (and stored in slots).

	Bytecode	Stack Variables	Copy Propagation
0	`iload_0`	`s0 = v0`	
1	`iload_1`	`s1 = v1`	
2	`iadd`	`s2 = s0 + s1`	
3	`istore_2`	`v2 = s2`	`v2 = v0 + v1`
4	`iload_2`	`s3 = v2`	
5	`ireturn`	`return s3`	`return v2`

We get from *bytecode* to *variables* by assigning identifiers to every value pushed or popped, e.g., iadd pops two operands to add and pushes the result.

We then apply a technique called *copy propagation* to eliminate some of the redundant variables. Copy propagation is a form of inlining in which references to variables are simply replaced with the assigned value, provided the transformation is valid.

Lee Benfield started out life on a Dragon 32 and hasn't looked back. He currently works on high-frequency trading systems, is interested in reverse engineering, and collects 8 bit computers to offset his Java memory footprint. Lee can be contacted at lee@benf.org

Mike Strobel studied Computer Science at Georgia Tech and works with a high-frequency trading group in Manhattan. He uses runtime code generation to alleviate the memory and throughput issues inherent in dynamic systems. Contact him at mike.strobel@gmail.com

Restoring variable names

If variables are reduced to slot references in the byte code, how then do we restore the original variable names? It's possible we can't. To improve the debugging experience, the byte code for each method may include a special section called the local variable table. For each variable in the original source, there exists an entry that specifies the name, the slot number, and the bytecode range for which the name applies. This table (and other useful metadata) can be included in the javap disassembly by including the `-l` option. For our `plus()` method above, the table `javap` emits looks like this:

Start	Length	Slot	Name	Signature
0	6	0	a	I
0	6	1	b	I
4	2	2	c	I

Here, we see that `v2` refers to an integer variable originally named `c` at bytecode offsets #4–5.

For classes that have been compiled without local variable tables (or which had them stripped out by an obfuscator), we have to generate our own names. There are many strategies for doing this; a clever implementation might look at how a variable is used for hints on an appropriate name.

What do we mean by 'valid'? Well, there are some important restrictions. Consider the following:

```
0: s0 = v1
1: v1 = s4
2: v2 = s0 <-- s0 cannot be replaced with v1
```

Here, if we were to replace `s0` with `v1`, the behaviour would change, as the value of `v1` changes after `s0` is assigned, but before it is consumed.

But what are the types?

When dealing with stack values, the JVM uses a more simplistic type system than Java source. Specifically, `boolean`, `char`, and `short` values are manipulated using the same instructions as `int` values. Thus, the comparison `v0 != 0` could be interpreted as:

```
v0 != false ? v1 : v2
```

or:

```
v0 != 0 ? v1 : v2
```

or even:

```
v0 != false ? v1 == true : v2 == true
```

and so on!

In this case, however, we are fortunate to know the exact type of `v0`, as it is contained within the method descriptor:

```
descriptor: (ZII)I
flags: ACC_PUBLIC, ACC_STATIC
```

This tells us the method signature is of the form:

```
public static int plus(boolean, int, int)
```

We can also infer that `v3` should be an `int` (as opposed to a `boolean`) because it is used as the return value, and the descriptor tells us the return type. We are then left with:

```
v3 =  v0 ? v1 : v2
return v3
```

As an aside, if `v0` were a local variable (and not a formal parameter) then we might not know it represents a `boolean` value and not an `int`. Remember that local variable table we mentioned earlier, which tells us the original variable names? It also includes information about the variable *types*, so if your compiler is configured to emit debug information, we can look to that table for type hints. There is another table called `LocalVariableTypeTable` [JVMb] that contains similar information; the key difference is that a `LocalVariableTypeTable` may include detailed information about generic types, whereas a `LocalVariableTable` cannot. It's worth noting that these tables are *unverified* metadata, so they ***are not necessarily trustworthy***. A particularly devious obfuscator might choose to fill these tables with lies, and the resulting bytecode would still be valid! Use them at your own discretion.

To avoid these complications, we only use copy propagation to inline variables that are assigned *exactly once*.

One way to enforce this might be to track all stores to non-stack variables, i.e., we know that `v1` is assigned `v10` at #0, and also `v11` at #2. Since there is more than one assignment to v1, we cannot perform copy propagation.

Our original example, however, has no such complications, and we end up with a nice, concise result:

```
v2 = v0 + v1
return v2
```

Stack analysis

In the previous example, we could guarantee which value was on top of the stack at any given point, and so we could name `s0`, `s1`, and so on.

So far, dealing with variables has been pretty straightforward because we've only explored methods with a single code path. In a real world application, most methods aren't going to be so accommodating. Each time you add a loop or condition to a method, you increase the number of possible paths a caller might take. Consider this modified version of our earlier example:

```
public static int plus(boolean t, int a, int b) {
  int c = t ? a : b;
  return c;
}
```

Now we have control flow to complicate things; if we try to perform the same assignments as before, we run into a problem.

	Bytecode	Stack Variables
0	`iload_0`	`s0 = v0`
1	`ifeq 8`	`if (s0 == 0) goto #8`
4	`iload_1`	`s1 = v1`
5	`goto 9`	`goto #9`
8	`iload_2`	`s2 = v2`
9	`istore_3`	`v3 = {s1,s2}`
10	`iload_3`	`s4 = v3`
11	`ireturn`	`return s4`

We need to be a little smarter about how we assign our stack identifiers. It's no longer sufficient to consider each instruction on its own; we need to keep track of how the stack looks at any given location, because there are multiple paths we might take to get there.

When we examine `#9`, we see that `istore_3` pops a single value, but that value has two sources: it might have originated at either `#5` or `#8`. The value at the top of the stack at `#9` might be either `s1` or `s2`, depending on whether we came from `#5` or `#8`, respectively. Therefore, we need to consider these to be the same variable – we merge them, and all references to `s1` or `s2` become references to the unambiguous variable `s{1,2}`. After this 'relabelling', we can safely perform copy propagation.

	After Relabelling	After Copy Propagation
0	`s0 = v0`	
1	`if (s0 == 0) goto #8`	`if (v0 == 0) goto #8`
4	`s{1,2} = v1`	`s{1,2} = v1`
5	`goto #9`	`goto #9`
8	`s{1,2} = :v2`	`s{1,2} = v2`
9	`v3 = s{1,2}`	`v3 = s{1,2}`
10	`s4 = v3`	
11	`return s4`	`return v3`

Notice the conditional branch at `#1`: if the value of `s0` is zero, we jump to the `else` block; otherwise, we continue along the current path. It's interesting to note that the test condition is negated when compared to the original source. We've now covered enough to dive into...

Conditional expressions

At this point, we can determine that our code could be modelled with a ternary operator (`?:`): we have a conditional, with each branch having a single assignment to the same stack variable `s{1,2}`, after which the two paths converge. Once we've identified this pattern, we can roll the ternary up straight away.

	After Copy Prop.	Collapse Ternary
0		
1	`if (v0 == 0) goto #8`	
4	`s{1,2} = v1`	
5	`goto 9`	
8	`s{1,2} = v2`	
9	`v3 = s{1,2}`	`v3 = v0 != 0 ? v1 : v2`
10		
11	`return v3`	`return v3`

Note that, as part of the transformation, we negated the condition at `#9`. It turns out that `javac` is fairly predictable in how it negates conditions, so we can more closely match the original source if we flip those conditions back.

Short-Circuit Operators ('&&' and '||')

```
public static boolean fn(boolean a,
                         boolean b, boolean c){
  return a || b && c;
}
```

How could anything be simpler? The bytecode is, unfortunately, a bit of a pain...

	Bytecode	Stack Variables	After Copy Propagation
0	`iload_0`	`s0 = v0`	
1	`ifne #12`	`if (s0 != 0) goto #12`	`if (v0 != 0) goto #12`
4	`iload_1`	`s1 = v1`	
5	`ifeq #16`	`if (s1 == 0) goto #16`	`if (v1 == 0) goto #16`
8	`iload_2`	`s2 = v2`	
9	`ifeq #16`	`if (s2 == 0) goto #16`	`if (v2 == 0) goto #16`
12	`iconst_1`	`s3 = 1`	`s{3,4} = 1`
13	`goto #17`	`goto 17`	`goto 17`
16	`iconst_0`	`s4 = 0`	`s{3,4} = 0`
17	`ireturn`	`return s{3,4}`	`return s{3,4}`

The `ireturn` instruction at `#17` could return either `s3` or `s4`, depending on what path is taken. We alias them as described above, and then we perform copy propagation to eliminate `s0`, `s1`, and `s2`.

We are left with three consecutive conditionals at `#1`, `#5` and `#7`. As we mentioned earlier, conditional branches either jump or fall through to the next instruction.

The bytecode above contains sequences of conditional branches that follow specific and very useful patterns:

Conditional Conjunction (&&)	Conditional Disjunction (||)

`T1:`
`  if (c1) goto L1`
`  if (c2) goto L2`
`L1:`
`  ...`
Becomes:
`  if (!c1 && c2) goto L2`
`L2:`
`  ...`
	`T1:`
	`  if (c1) goto L2`
	`  if (c2) goto L2`
	`L1:`
	`  ...`
	Becomes:
	`  if (c1 || c2) goto L2`
	`L1:`
	`  ...`

If we consider neighbouring pairs of conditionals above, `#1`...`#5` do not conform to either of these patterns, but `#5`...`#9` is a conditional disjunction (`||`), so we apply the appropriate transformation:

```
 1:  if (v0 != 0) goto #12
 5:  if (v1 == 0 || v2 == 0) goto #16
12:  s{3,4} = 1
13:  goto #17
16:  s{3,4} = 0
17:  return s{3,4}
```

Note that every transform we apply might create opportunities to perform *additional* transforms. In this case, applying the `||` transform restructured our conditions, and now `#1`...`#5` conform to the `&&` pattern! Thus, we can further simplify the method by combining these lines into a single conditional branch:

```
 1:  if (v0 == 0 && (v1 == 0 || v2 == 0)) goto #16
12:  s{3,4} = 1
13:  goto #17
16:  s{3,4} = 0
17:  return s{3,4}
```

Does this look familiar? It should: this bytecode now conforms to the ternary (`?:`) operator pattern we covered earlier. We can reduce `#1`...`#16` to a single expression, then use copy propagation to inline `s{3,4}` into the `return` statement at `#17`:

```
return (v0 == 0 && (v1 == 0 || v2 == 0)) ? 0 : 1;
```

Using the method descriptor and local variable type tables described earlier, we can infer all the types necessary to reduce this expression to:

```
return (v0 == false && (
        v1 == false || v2 == false))
        ? false : true;
```

Well, this is certainly more concise than our original decompilation, but it's still pretty jarring. Let's see what we can do about that. We can start by collapsing comparisons like `x == true` and `x == false` to `x` and `!x`, respectively. We can also eliminate the ternary operator by reducing `x ? false : true` as the simple expression `!x`.

```
return !(!v0 && (!v1 || !v2));
```

Better, but it's still a bit of a handful. If you remember your high school discrete maths, you can see that De Morgan's theorem can be applied here:

```
!(a || b) --> (!a) && (!b)
!(a && b) --> (!a) || (!b)
```

And thus:

```
return ! ( !v0 && ( !v1 || !v2 ) )
```

becomes:

```
return ( v0 || !(!v1 || !v2 ) )
```

and eventually:

```
return ( v0 || (v1 && v2) )
```

Hurrah!

Dealing with method calls

We've already seen what it looks like for a method to be called: arguments 'arrive' in the locals array. To *call* a method, arguments must be pushed onto the stack, following a `this` pointer in the case of instance methods). Calling a method in bytecode is as you'd expect:

```
push arg_0
push arg_1
invokevirtual METHODREF
```

We've specified `invokevirtual` above, which is the instruction used to invoke most instance methods. The JVM actually has a handful of instructions used for method calls, each with different semantics:

1. `invokeinterface` invokes interface methods.
2. `invokevirtual` invokes instance methods using virtual semantics, i.e., the call is dispatched to the proper override based on the runtime type of the target.

3. **`invokespecial`** invokes a specific instance method (without virtual semantics); it is most commonly used for invoking constructors, but is also used for calls like **`super.method()`**.
4. **`invokestatic`** invokes static methods.
5. **`invokedynamic`** is the least common (in Java), and it uses a 'bootstrap' method to invoke a custom call site binder. It was created to improve support for dynamic languages, and has been used in Java 8 to implement lambdas.

The important detail for a decompiler writer is that the class's *constant pool* [JVMc] contains details for any method called, including the number and type of its arguments, as well as its return type. Recording this information in the caller class allows the runtime to verify that the intended method exists at runtime, and that it conforms to the expected signature. If the target method is in third party code, and its signature changes, then any code that attempts to call the old version will throw an error (as opposed to producing undefined behaviour).

Going back to our example above, the presence of the **`invokevirtual`** opcode tells us that the target method is an *instance method*, and therefore requires a **`this`** pointer as an implicit first argument. The **`METHODREF`** in the constant pool tells us that the method has one formal parameter, so we know we need to pop an argument off of the stack in addition to the target instance pointer. We can then rewrite the code as:

```
arg_0.METHODREF(arg_1)
```

Of course, bytecode isn't always so friendly; there's no requirement that stack arguments be pushed neatly onto the stack, one after the other. If, for example, one of the arguments was a ternary expression, then there would be intermediate load, store, and branch instructions that will need to be transformed independently. Obfuscators might rewrite methods into a particularly convoluted sequence of instructions. A good decompiler will need to be flexible enough to handle many interesting edge cases that are beyond the scope of this article.

There has to be more to it than this...

So far, we've limited ourselves to analysing a single sequence of code, beginning with a list of simple instructions and applying transformations that produce more familiar, higher-level constructs. If you are thinking that this seems a little too simplistic, then you are correct. Java is a highly structured language with concepts like scopes and blocks, as well as more complicated control flow mechanisms. In order to deal with constructs like **`if`**/**`else`** blocks and loops, we need to perform a more rigorous analysis of the code, paying special attention to the various paths that might be taken. This is called *control flow analysis*.

We begin by breaking down our code into contiguous blocks that are guaranteed to execute from beginning to end. These are called *basic blocks*, and we construct them by splitting up our instruction list along places where we might jump to another block, as well as any instructions which might be jump targets themselves.

We then build up a control flow graph (CFG) by creating edges between the blocks to represent all possible branches. Note that these edges may not be *explicit* branches; blocks containing instructions that might throw exceptions will be connected to their respective exception handlers. We won't go into detail about constructing CFGs, but some high level knowledge is required to understand how we use these graphs to detect code constructs like loops.

Figure 1 shows some examples of control flow graphs.

The aspects of control flow graphs that we are most interested in are domination relationships:

- A node **`D`** is said to *dominate* another node **`N`** if all paths to **`N`** pass through **`D`**. All nodes dominate themselves; if **`D`** and **`N`** are different nodes, then **`D`** is said to *strictly dominate* **`N`**.
- If **`D`** strictly dominates **`N`** and does not strictly dominate any *other* node that strictly dominates **`N`**, then **`D`** can be said to *immediately dominate* **`N`**.

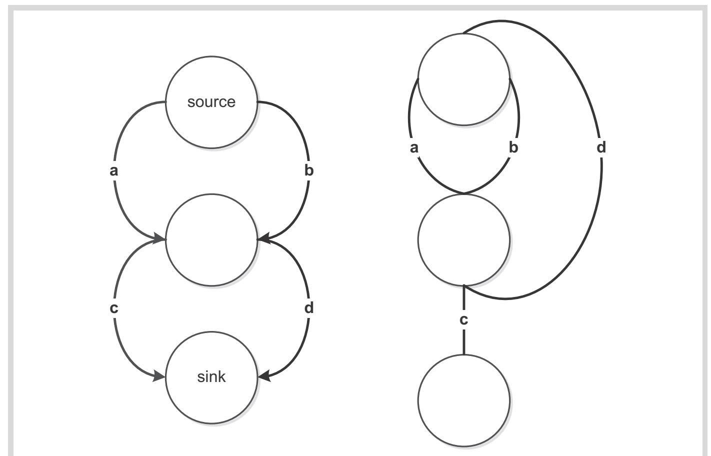


Figure 1

- The *dominator tree* is a tree of nodes in which each node's children are the nodes it immediately dominates.
- The *dominance frontier* of **`D`** is the set of all nodes **`N`** such that **`D`** dominates an immediate predecessor of **`N`** but does not strictly dominate **`N`**. In other worse, it is the set of nodes where **`D`**'s dominance ends.

Basic loops and control flow

Consider the following Java method:

```
public static void fn(int n) {
  for (int i = 0; i < n; ++i) {
    System.out.println(i);
  }
}
```

and its disassembly:

```
 0:  iconst_0
 1:  istore_1
 2:  iload_1
 3:  iload_0
 4:  if_icmpge 20
 7:  getstatic #2       // System.out:PrintStream
10:  iload_1
11:  invokevirtual #3   // PrintStream.println:(I)V
14:  iinc 1, 1
17:  goto 2
20:  return
```

Let's apply what we discussed above to convert this into a more readable form, first by introducing stack variables, then performing copy propagation.

	Bytecode	Stack Variables	After Copy Propagation
0	`iconst_0`	`s0 = 0`	
1	`istore_1`	`v1 = s0`	`v1 = 0`
2	`iload_1`	`s2 = v1`	
3	`iload_0`	`s3 = v0`	
4	`if_icmpge 20`	`if (s2 >= s3) goto 20`	`if (v1 >= v0) goto 20`
7	`getstatic #2`	`s4 = System.out`	
10	`iload_1`	`s5 = v1`	
11	`invokevirtual #3`	`s4.println(s5)`	`System.out.println(v1)`
14	`iinc 1, 1`	`v1 = v1 + 1`	`v1 = v1 + 1`
17	`goto 2`	`goto 2`	`goto 4`
20	`return`	`return`	`return`

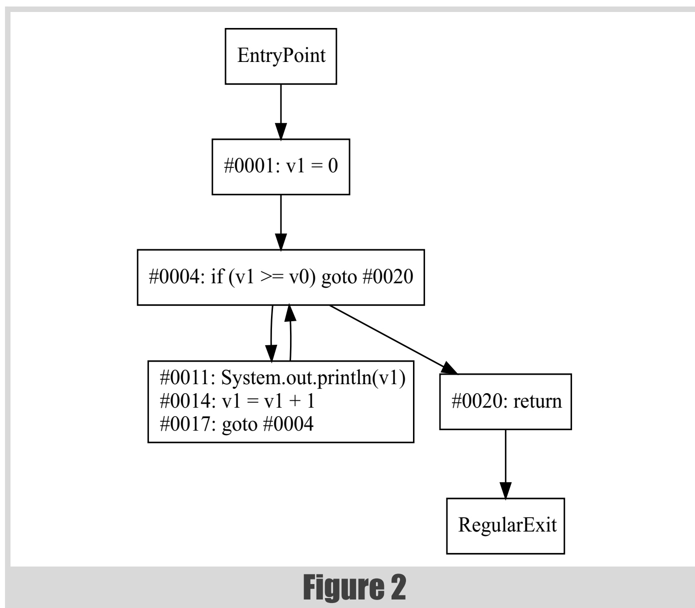


Figure 2

Notice how the conditional branch at **#4** and the **goto** at **#17** create a logical loop. We can see this more easily by looking at a control flow graph (Figure 2).

From the graph, it's obvious that we have a neat cycle, with an edge from the **goto** back to a conditional branch. In this case, the conditional branch is referred to as a *loop header*, which can be defined as a dominator with a loop-forming backward edge. A loop header dominates all nodes within the loop's body.

We can determine whether a condition is a loop header by looking for a loop-forming back edge, but how do we do that? A simple solution is to test whether the condition node is in its own dominance frontier. Once we know we have a loop header, we have to figure out which nodes to pull into the loop body; we can do that by finding all nodes dominated by the header. In pseudocode, our algorithm would look something like Listing 2.

Once we've figured out the loop body, we can transform our code into a loop. Keep in mind that our loop header may be a conditional jump *out* of the loop, in which case we need to negate the condition.

```
v1 = 0
while (v1 < v0) {
  System.out.println(v1)
  v1 = v1 + 1
}
return
```

And voilà, we have a simple pre-condition loop! Most loops, including **while**, **for** and **for-each**, compile down to the same basic pattern, which we detect as a simple **while** loop. There's no way to know for sure what kind of loop the programmer originally wrote, but **for** and **for-each** follow pretty specific patterns that we can look for. We won't go into the details, but if you look at the **while** loop above, you can see how the original **for** loop's initializer (**v1 = 0**) precedes the loop, and its iterator (**v1 = v1 + 1**) has been inserted at the end of the loop body. We'll leave it as an exercise to think about when and how one might transform **while** loops into **for** or **for-each**. It's also interesting to think about how we might have to adjust our logic to detect *post-conditional* (**do**/**while**) loops.

```
findDominatedNodes(header)
    q := new Queue()
    r := new Set()

    q.enqueue(header)

    while (not q.empty())
        n := q.dequeue()

        if (header.dominates(n))
            r.add(n)

            for (s in n.successors())
                q.enqueue(n)

    return r
```

Listing 2

We can apply a similar technique to decompile **if**/**else** statements. The bytecode pattern is pretty straightforward:

```
begin:
  iftrue(!condition) goto #else
  // 'if' block begins here
  ...
  goto #end
else:
  // 'else' block begins here
  ...
end:
  // end of 'if/else'
```

Here, we use **iftrue** as a pseudo-instruction to represent any conditional branch: test a condition, and if it passes, then branch; otherwise, continue. We know that the **if** block starts at the instruction following the condition, and the **else** block starts at the condition's jump target. Finding the contents of those blocks is as simple as finding the nodes dominated by those starting nodes, which we can do using the same algorithm we described above.

We've now covered basic control flow mechanisms, and while there are others (e.g., exception handlers and subroutines), those are beyond the scope of this introductory article.

Wrap-up

Writing a decompiler is no simple task, and the experience easily translates into a book's worth of material – perhaps even a series of books! Obviously, we could not cover everything in a single article, and you probably wouldn't want to read it if we'd tried. We hope that by touching on the most common constructs – logical operators, conditionals, and basic control flow – we have given you an interesting glimpse into the world of decompiler development.

Now go write your own! :) ■

References

This article originally appeared as a Blog post in the Java Advent Calendar, and is released under the Creative Commons 3.0 Attribution license.

(In general, just read the JVM spec end to end!)

[JVMa] JVM Specification §2.6.2 – http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.6.2

[JVMb] JVM Specification §4.7.14 – http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.7.14

[JVMc] JVM Specification §4.4 – http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4

Optimizing Big Number Arithmetic Without SSE

Addition and multiplication can be slow for big numbers. Sergey Ignatchenko and Dmytro Ivanchykhin try to speed things up.

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

There is one thing which has puzzled me for a while, and it is the performance of programs written in C when it comes to big numbers. It may or may not help with the decades-long 'C vs Fortran' performance debate, but let's concentrate on one single and reasonably well-defined thing – big number arithmetic in C and see if it can be improved.

In fact, there are very few things which gain from being rewritten in assembler (compared to C), but big number arithmetic is one of them, with relatively little progress in this direction over the years. Let's take a look at OpenSSL (a library which is among the most concerned about big number performance: 99% of SSL connections use RSA these days, and RSA_performance == Big_Number_Performance, and RSA is notoriously ssllloooowww). Run:

```
openssl speed rsa
```

(if you're on Windows, you may need to install OpenSSL first) and see what it shows. In most cases, at the end of the benchmark report it will show something like

```
compiler: gcc -fPIC -DOPENSSL_PIC -DZLIB
-DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN
-DHAVE_DLFCN_H -DKRB5_MIT -DL_ENDIAN -DTERMIO -Wall
-O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2
-fexceptions -fstack-protector --param=ssp-buffer-
size=4 -m32 -march=i686 -mtune=atom -fasynchronous-
unwind-tables -Wa,--noexecstack
-DOPENSSL_BN_ASM_PART_WORDS -DOPENSSL_IA32_SSE2
-DOPENSSL_BN_ASM_MONT -DSHA1_ASM -DSHA256_ASM
-DSHA512_ASM -DMD5_ASM -DRMD160_ASM -DAES_ASM
-DWHIRLPOOL_ASM
```

Note those `-DOPENSSL_BN_ASM` in the output – it means that OpenSSL prefers to use assembler for big number calculations. It was the case back in 1998, and it is still the case now (last time I checked, the difference between C and asm implementations was 2x, but it was long ago, so things may have easily changed since). But why should this be the case? Why with all the optimizations compilers are doing now, should such a common and trivial thing still need to be written in asm (which has its own drawbacks – from the need to write it manually for each and every platform, to sub-optimality of generic asm when it comes to pipeline optimizations – and hand-rewriting asm for each new CPU -march/-mtune is not realistic)? If it can perform in asm but cannot perform in C, it means that all hardware support is present, but the performance is lost somewhere in between C developer and generated binary code; in short – the compiler cannot produce good code. This article will try to perform some analysis of this phenomenon.

The problem

For the purposes of this article, let's restrict our analysis to the two most basic operations in big-number arithmetic: addition and multiplication. Also in this article we won't go into SSE/SSE2, concentrating on much simpler things which still can provide substantial performance gains. The reason why I decided to stay away from SSE/SSE2/... is because while the whole area of vectorization has been extensively studied recently, it seems that people have forgotten about the good old plain instruction set, which is still capable of delivering good performance; combined with the fact that SSE has its own issues which make it not so optimal in certain scenarios, and another suggestion that the most optimal implementation would probably parallelise execution between the SSE engine and the good old plain instruction set.

Mind the gap

When I'm speaking about addition: sure, C/C++ has operator **+**. However, to add, say, two 128-bit numbers, we need to do more than just repeating twice an addition of two `uint64_t`'s in the C/C++ sense, we also need to obtain carry (65th bit of result) which may easily occur when adding lower 64-bit parts of our operands.

```
uint128_t c = (uint128_t)a + (uint128_t)b;
```

can be rewritten as

```
pair<bool,uint64_t> cLo =
  add65(lower64(a) + lower64(b));
uint64_t cHi = higher64(a)+higher64(b)+cLo.first;
```

(assuming that `lower64()` returns low 64 bits, `higher64()` returns high 64 bits, and `add65()` returns a pair representing `<carry flag,64_bits_of_sum>`)

It is this carry flag which is causing all the trouble with additions. In x64 assembler, this whole thing can be written as something like

```
add reg_holding_lower64_of_a,
  reg_holding_lower64_of_b
adc reg_holding_higher64_of_a,
  reg_holding_higher64_of_b
```

– and that's it. However, as in C there is no such thing as a 'carry flag', developers are doing all kinds of stuff to simulate it, usually with pretty bad results for performance.

'No Bugs' Bunny 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 is currently holding the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

Dmytro Ivanchykhin has 10+ years of development experience, and has a strong mathematical background (in the past, he taught maths at NDSU in the United States). Dmytro can be contacted at d_ivanchykhin@yahoo.com

there is a substantial gap between the capabilities of the hardware and the capabilities of the C

With multiplication it is also not that easy, and for a similar reason. In mathematics, when multiplying 64 bits by 64 bits, the result is 128-bit long. In assembler (at least in x64, and also probably in ARM) the result is also 128-bit long. However, in C, the result of 64bit-by-64-bit multiplication is 64-bit long, which, when working with big number arithmetic, leads to four-fold (!) increase in number of multiplications. Ouch.

It should be noted that at least on x64 there is an instruction which multiplies 64 bits by 64 bits and returns lower 64 bits of the result, and this instruction is somewhat faster than full-scale 64-bit-by-64-bit-returning-128-bit multiplication. Still, as we'll see, it is not enough to compensate for the four-fold increase in number of multiplications described above.

Overall, it looks that there is a substantial gap between the capabilities of the hardware and the capabilities of the C. The whole thing seems to be one incarnation of the Over-Generic Use of Abstractions [NoBugs11] which, as it often happens, leads to a waste of resources.[1] However, the question is not exactly "Who is to blame?" [Herzen45], but "What Is to Be Done?" [Chernyshevsky63].

So far, so bad

So, what can be done in this regard? I've tried to perform some very basic experiments (based on things which were rather well-known some time ago, but apparently forgotten now) trying to implement `uint256_t` keeping in mind considerations outlined above, and to compare its performance with the performance of `boost::uint256_t` under recent versions of both MSVC and GCC (see the 'Benchmark' section for details). The results I've managed to obtain show between 1.8x and 6.3x improvements over `boost::uint256_t` (both for addition and for multiplication), which, as I feel, indicate a step in the right direction.

Addition

To simplify the description, I'll use `uint128_t` as an example; however, the same approach can be easily generalized to any other types (including `uint256_t` which I've used for benchmarks).

One thing which I've tried to do to deal with addition-with-carry, was:

```
struct myuint128_t { uint64_t lo; uint64_t hi; };
inline my_uint128_t add128(my_uint128_t a,
                           my_uint128_t b) {
  my_uint128_t ret;
  ret.cLo = a.lo + b.lo;
  ret.cHi = a.hi + b.hi + (ret.cLo<b.lo);
  return ret;
}
```

This is mathematically equivalent to the 128-bit addition, and it (with subtle differences depending on compiler, etc.) indeed compiles into something like:

```
mov rax, a.lo

add rax, b.lo //got cLo in rax
cmp rax, b.lo //setting carry flag to
              //"ret.cLo < b.lo" mov rbx, a.hi
sbb rdx,rdx   //!
neg rdx       //!
add rbx, rdx
add rbx, b.hi
```

Compilers could do significantly better if they would remove two lines marked with `(!)`, and replace the `add` in the next line with `adc` (lines marked with `'!'` move carry to `rdx`, which is then added to addition of `rbx`, `rdx` – `adc` will do the same thing and faster). Actually, compilers can go further and optimize away `cmp rax, b.lo` too (on the grounds that carry flag has already been set in the previous line). After these optimizations, the code would look as follows:

```
mov rax, a.lo
add rax, b.lo //got cLo in rax
mov rbx, a.hi
adc rbx, b.hi
```

This looks as a perfect asm for our 128-bit addition, doesn't it? And from this point, compiler can optimize it from a pipelining optimization point of view to its heart's content.

It is interesting to note that despite all the unnecessary stuff, our C still performs reasonably well compared to `boost::`.

`boost::` uses an alternative approach, splitting `uint128_t` into four `uint32_t`'s and performing four additions; as my tests show, it seems to be slower (and I also feel that optimizing my code should be easier for compiler writers – as outlined above, only two rather relatively minor optimizations are necessary to speed multi-word addition to the 'ideal' level).

A word of caution: while this "a.hi + b.hi + (cLo<b.lo)" technique seems to be handled reasonably optimally by most modern compilers, when trying to compile my code which uses `uint64_t` for 32-bit platforms, with both GCC and MSVC, they tend to generate very ugly code (with conditional jumps, pipeline stalls etc.); while I didn't see such behaviour in any other scenario – I cannot be 100% sure that it is the only case when the compiler goes crazy. When compiling for 32-bit platforms, one may use similar technique, but using natively supported `uint32_t`'s instead of simulated `uint64_t`'s.

Multiplication

With multiplication, things tend to get significantly more complicated. To write 64-by-64 multiplication in bare C (and without `uint128_t` which is not natively supported by most of the compilers), one would need to write something like Listing 1.

However, in most cases we can access 64-bit-by-64-bit-returning-128-bit multiplication instruction (which, as I've seen, is available on most CPUs in modern use). In particular, on MSVC there is an 'intrinsic' `_umul128`,

1. Here an abstraction, which I see as being over-generic, is an abstraction of C operators, which often return the result of the same size as the size of its arguments

```
inline my_uint128_t mul64x64to128(uint64_t a,
                                  uint64_t b) {
  my_uint128_t ret;
  uint64_t t0 = ((uint64_t)(uint32_t)a) *
                ((uint64_t)(uint32_t)b);
  uint64_t t1 = (a>>32)*((uint64_t)(uint32_t)b) +
                (t0>>32);
  uint64_t t2 = (b>>32)*((uint64_t)(uint32_t)a) +
                ((uint32_t)t1);
  ret.lo = (((uint64_t)((uint32_t)t2))<<32) +
                ((uint32_t)t0);
  ret.hi = (a>>32) * (b>>32);
  ret.hi += (t2>>32) + (t1>>32);
  return ret;
}
```

Listing 1

which does exactly what we need. Then, our multiplication of `uint64_t`'s can be rewritten as:

```
inline my_uint128_t mul64x64to128(uint64_t a,
                                  uint64_t b) {
  my_uint128_t ret;
  ret.lo = _umul128(a, b, &(ret.hi));
  return ret;
}
```

In fact, a pure-C/C++ (without intrinsics) implementation uses 6 'type A' (64-bit-by-64-bit-returning-64-bit) multiplications, and an intrinsic-based one uses 3 multiplications, two being 'type A' and one being 'type B' (64-bit-by-64-bit-returning-128-bit). However, the difference in speed between the two is less than 2x, and I tend to attribute it to 'type A' multiplications being faster than 'type B' ones, at least on x64 CPUs I've tested my programs on. Still, intrinsic-based version looks significantly faster (than both my own pure-C implementation, and than `boost::` pure-C implementation).

Under GCC, there is no such intrinsic, but it can be simulated using the following 5-line GCC inline asm:

```
void multiply(uint64_t& rhi, uint64_t& rlo,
              uint64_t a, uint64_t b)
{
  __asm__(
  "    mulq  %[b]\n"
  :"=d"(rhi),"=a"(rlo)
  :"1"(a),[b]"rm"(b));
}
```

This one (from [Crypto++]) works, too:

```
#define MultiplyWordsLoHi(p0, p1, a, b)  asm  \
("mulq %3" : "=a"(p0), "=d"(p1) : "a"(a),     \
"r"(b) : "cc");
```

Benchmarks

Benchmark results (benchmarking was performed for 256-bit integers, but the same approach can be generalized to big integers of any size; also, where applicable, boost version 1.55.0 has been used):

	Visual Studio 2013	GCC 4.8.1
Addition (`boost::uint256_t`)	0.0202 op/clock	0.0202 op/clock
Addition (`uint256_t` according to present article)	0.0366 op/clock	0.1282 op/clock
Advantage over `boost::uint256_t`	1.83x	6.34x
Multiplication (`boost::uint256_t`)	0.0160 op/clock	0.0197op/clock
Multiplication (`uint256_t` according to present article)	0.0285 op/clock	0.0549 op/clock
Advantage over `boost::uint256_t`	1.78x	2.79x

It should be noted that absolute 'op/clock' numbers should not be used for comparison between different computers (let alone comparison between different architectures/compilers).

Conclusion

We've taken a look at problems with big number arithmetic in C, and described some optimizations which may speed things up significantly, providing somewhere around 2x to 6x gains over the current `boost::` implementation.

Also note that while all examples were provided for x64 asm, very similar problems (and very similar solutions) seem to apply also to x86 and to ARM (both these platforms have both "add with carry" and "multiply_with_double_width_result" asm-level operations, and these two operations are the only things necessary to deal with the big number arithmetic reasonably efficiently). ■

References

[Chernyshevsky63] Nikolai Chernyshevsky, *What Is to Be Done?*, 1863

[Crypto++] http://www.cryptopp.com/docs/ref/integer_8cpp_source.html

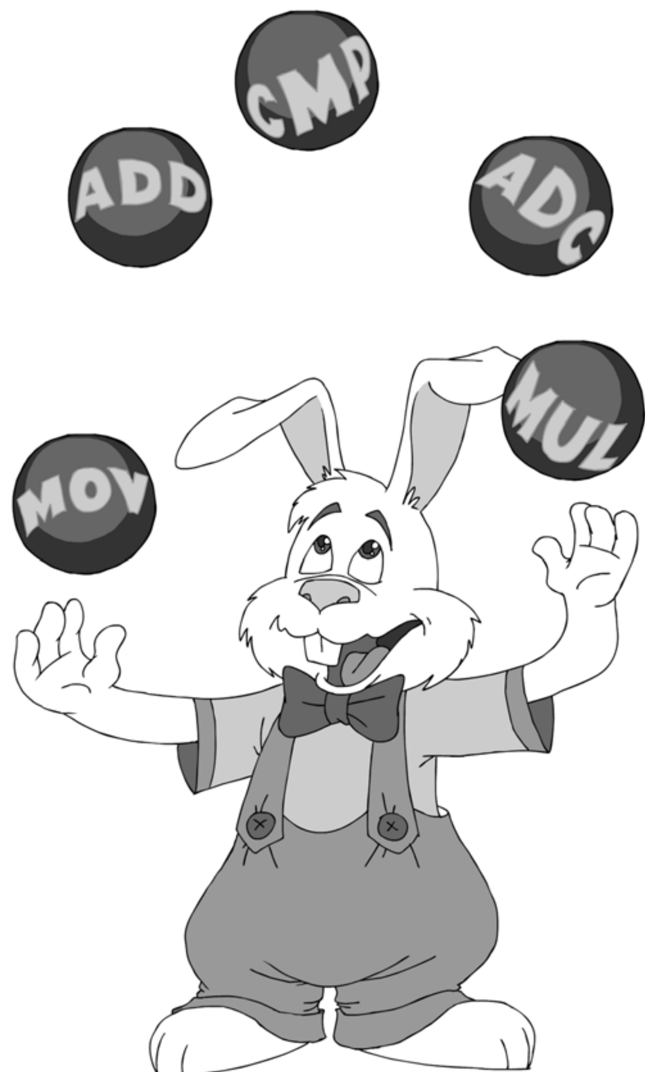
[Herzen45] Alexander Herzen, *Who is to Blame?*, 1845–1846

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

[NoBugs11] Sergey Ignatchenko. 'Over-generic use of abstractions as a major cause of wasting resources'. *Overload*, August 2011

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.



Capturing lvalue References in C++11 Lambdas

How confusing does it get when references refer to references and references are captured by value? Pete Barber shows us that it all falls out in the C++ consistency wash.

Recently the question 'what is the type of an lvalue reference when captured by reference in a C++11 lambda?' was raised at work. It turns out that it's a reference to whatever the original reference was too. This is just like taking a reference to an existing reference, e.g.

```
int foo = 7;
int& rfoo = foo;
int& rfoo1 = rfoo;
int& rfoo2 = rfoo1;
```

All references refer to `foo` rather than `rfoo2->rfoo1->rfoo->foo` which means running the code in Listing 1, gives the following results:

```
foo:7, rfoo:7, rfoo1:7, rfoo2:7
foo:8, rfoo:8, rfoo1:8, rfoo2:8
&foo:00D3FB0C, &rfoo:00D3FB0C,
&rfoo1:00D3FB0C,   &rfoo2:00D3FB0C
```

I.e. all the references are aliases for the original `foo` hence the same value is displayed including when the original is modified and that the address of each variable is the same, that of `foo`.

There is nothing surprising here. It's just basic C++ but it's a long time since I've thought about it which is why with lambdas, l-value, r-value and universal references I sometimes do a double take on what was once obvious.

The same happens with lambda capture but it's a slightly more interesting story. Take the example in Listing 2, which gives:

```
foo:99, rfoo:99, rfoo1:99
&foo:00D3FB0C, &rfoo:00D3FB0C, &rfoo1:00D3FB0C
foo:99
rfoo:99
rfoo1:100
&foo:00D3FAE0, &rfoo:00D3FAE4, &rfoo1:00D3FB0C
```

To begin with it behaves as per the first example in that `foo`, `rfoo` and `rfoo1` all give the same value. This is because `rfoo` and `rfoo1` are effectively aliases for `foo` as shown when displaying their addresses; they're all the same.

```
std::cout << "foo:" << foo << ", rfoo:" << rfoo
          << ", rfoo1:" << rfoo1 << ", rfoo2:"
          << rfoo2
          << '\n';
++foo;

std::cout << "foo:" << foo << ", rfoo:" << rfoo
          << ", rfoo1:" << rfoo1 << ", rfoo2:"
          << rfoo2
          << '\n';

std::cout << "&foo:" << &foo << ", &rfoo:"
          << &rfoo  << ", &rfoo1:" << &rfoo1
          << ", &rfoo2:" << &rfoo2
          << '\n';
```

Listing 1

```
int foo = 99;
int& rfoo = foo;
int& rfoo1 = foo;

std::cout << "foo:" << foo << ", rfoo:" << rfoo
          << ", rfoo1:" << rfoo1
          << '\n';

std::cout << "&foo:" << &foo << ", &rfoo:"
          << &rfoo << ", &rfoo1:" << &rfoo1
          << '\n';

auto l = [foo, rfoo, &rfoo1]()
{
    std::cout << "foo:" << foo << '\n';
    std::cout << "rfoo:" << rfoo << '\n';
    std::cout << "rfoo1:" << rfoo1 << '\n';

    std::cout << "&foo:" << &foo << ", &rfoo:"
              << &rfoo << ", &rfoo1:" << &rfoo1
              << '\n';
};

foo = 100;

l();
```

Listing 2

However, when these same variables are captured it's a different story: The capture of `foo` is of no surprise as this is by-value so displays the captured value of 99 despite the original `foo` being changed to 100 prior to the lambda being invoked. Its address is that of a new variable; a member of the lambda.

It starts to get interesting with the capture of `rfoo`. When the lambda is invoked this too displays 99, the original captured value. Also, its address is not that of the original `foo`. It seems that the reference itself has not been captured but rather what it refers too, in this case an `int` with the value of 99. It appears to have been magically dereferenced as part of the capture.

This is the correct behaviour and when thought about becomes somewhat obvious. It's just like assigning a variable from a reference, e.g.

Pete Barber has been programming in C++ before templates and exceptions, C# from early on and BASIC before then. After working on backup and archiving products on Windows and UNIX, he's now writing mobile games along side Candy Crush Saga.He still programs for fun and occasionally writes about it at petebarber.blogspot.com. He can be contacted at pete.barber@gmail.com

```
int foo = 7;
int& rfoo = foo;
int bar = rfoo;
```

bar doesn't become an **int&** and **rfoo** is magically dereferenced except in this scenario there is nothing magical at all, it's as expected. If **int** were replaced with auto, e.g.

```
auto bar = rfoo;
```

then it would be expected that **bar** is an **int** as **auto** strips of CV and reference qualifiers.

Finally, there is **rfoo1**. This too is odd as it is attempting to take a reference to a reference. As seen in the first example this is perfectly fine. The end effect is that there can't be a reference to reference and so on and all are aliases of the original variable.

This is pretty much what's happening here. It's irrelevant that the target of the capture is a reference. In the end the capture by reference is capture by reference of the underlying variable, i.e. what **rfoo1** refers too, in this case **foo** not **rfoo1** itself. This is demonstrated twofold by **rfoo1** within the lambda displaying the updated value of **foo** and also that the address of **rfoo1** within the lambda is that of **foo** outside it.

This is as per the standard section 5.1.2 Lambda expression sub-note 14:

> An entity is captured by copy if it is implicitly captured and the capture-default is = or if it is explicitly captured with a capture that does not include an **&**. For each entity captured by copy, an unnamed nonstatic data member is declared in the closure type. The declaration order of these members is unspecified.
>
> **The type of such a data member is the type of the corresponding captured entity if the entity is not a reference to an object, or the referenced type otherwise.** [Note: If the captured entity is a reference to a function, the corresponding data member is also a reference to a function.]

The sentence in bold states that for a reference captured by value then the type of the captured value is the type referred to, i.e. the reference aspect has been removed the crucial part being 'or the referenced type otherwise'.[1]

Finally, Listing 3 is a vivid example showing that a reference captured by value involves a dereference.

The class **bar** provides a crude copy-constructor that sets the stored value to 9999. The following output is similar to that in the previous example in that the addresses of **bar** and **rbar** in the lambda differ from that of **bar** showing they're copies whilst **rbar1** is the same. Secondly, the value of **mValue** stored within **Bar** is shown as 9999 for the first two captured variables meaning they were copy-constructed.

```
&bar:00D3FB0C, &rbar:00D3FB0C, &rbar1:00D3FB0C
bar:9999
rbar:9999
rbar1:2
&bar:00D3FAE0, &rbar:00D3FAE4, &rbar1:00D3FB0C
```

Making the copy-construct private (by commenting out the seemingly unnecessary **public:**) prevents compilation. (See Listing 4.)

```
class Bar
{
private:
  int mValue;

public:
  Bar(const Bar&) : mValue(9999)
  {
  }

public:
  Bar(const int value) : mValue(value) {}
  int GetValue() const { return mValue; }
  void SetValue(const int value) {
    mValue = value; }
};

Bar bar(1);
Bar& rbar = bar;
Bar& rbar1 = bar;

std::cout << "&bar:" << &bar << ", &rbar:"
  << &rbar<< ", &rbar1:" << &rbar1 << '\n';

auto l2 = [bar, rbar, &rbar1]()
{
 std::cout << "bar:" << bar.GetValue() << '\n';
 std::cout << "rbar:" << rbar.GetValue() << '\n';
 std::cout << "rbar1:" << rbar1.GetValue()
           << '\n';

  std::cout << "&bar:" << &bar << ", &rbar:"
            << &rbar<< ", &rbar1:" << &rbar1
            << '\n';
};

bar.SetValue(2);

l2();
```

Listing 3

At first the whole capturing of references by reference seems somewhat mind bending and a unique issue. However, when briefly analysed it quickly becomes clear that there is nothing extraordinary happening at all. In fact it is pleasing to see that far from being complicated it is just another example of where references to references have to be considered and that their treatment in this context is the same as in others. The same is true for the capture of references by value. Consistency is good. ■

```
1>------ Build started: Project: References, Configuration: Debug Win32 ------
1>       main.cpp
1>       c:\users\pete\desktop\references\references\main.cpp(85): error C2248: 'Bar::Bar' : cannot
access private member declared in class 'Bar'
1>       c:\users\pete\desktop\references\references\main.cpp(59) : see declaration of 'Bar::Bar'
1>       c:\users\pete\desktop\references\references\main.cpp(54) : see declaration of 'Bar'
1>       c:\users\pete\desktop\references\references\main.cpp(59) : see declaration of 'Bar::Bar'
1>       c:\users\pete\desktop\references\references\main.cpp(54) : see declaration of 'Bar'
```

Listing 4

1. I haven't experimented with references to functions