overload 95

FEBRUARY 2010 £3

Quality Matters

We consider high quality mechanisms to diagnose problems in software

Simplifying the C++/Angelscript Binding Process

When languages collide: how to simplify C++ and Angelscript interoperability

The Model Student

We play a game of six integers

One Approach to Using Hardware Registers in C++

Unit testing increases software reliability. Martin Moene presents a technique for checking the control of physical hardware registers.

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 95

February 2010

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 96 should be submitted by 1st March 2010 and for Overload 97 by 1st May 2010.

ACCU

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

Copyrights and Trade Marks

Back To The Future

The last decade has seen huge changes. Ric Parkin looks at technology and its effects.

Welcome to the first Overload of the new decade! (Note: as defining a decade is purely a matter of semantics, I've decided that the common convention that the years starting with the same three digits – eg 200x and 201x – are a reasonable way of partitioning years into decades, so no pedantic letters please!) So on this completely arbitrary cusp, I thought I'd have a look back at what has changed technologically over the last decade, and peer into a foggy crystal ball to hazard a guess at what the next may bring.

This is going to be full of facts and figures, but so many I can't realistically give references without going way over the top. Most were found by search for 'history of X', if you're interested.

Disco 2000

So let's start with personal computers. In many ways they were quite similar to now, but as you would expect most of the parts were much less powerful: the latest chips were things like the Intel Pentium III, with around 9.5 million transistors, 512KB onboard cache, running at around 750MHz (although this was rising fast from 500MHz shortly before to 1MHz quite soon after). Windows 2000 was just about to be released, so most people were on Windows 98 or NT4, while Apple had been making a comeback with the iMac for a couple of years, and had just released OSX (but only for servers – the desktop version was still over a year away). Linux (and other free software such as StarOffice, which would shortly become OpenOffice) was increasingly being seen as a challenge to Microsoft's dominance of PC Operating Systems and Office software.

Most PCs came in big towers, the occasional 'pizza box', or still-bulky laptops, although more slimline desktops were making inroads. Monitors were pretty much exclusively big bulky CRTs with limited resolutions – the first 36" 1920×1200 monitors only appeared around 2000. Hard disk size was in the region of a few GB. USB2 was new but taking off, 3.5inch floppy disk drives were around but being phased out, replaced by CDs, USB sticks and file transfer via networks.

Handheld computing was still in its infancy in many ways. Small or tablet computers had been around for ages, notably the Apple Newton in the mid 90s, and Psion and Palm had popular PDAs, but they had never really sold beyond some business use and and tech hobbyists. Mobile phones, while they had become a mainstream device during the 90s, were still mainly simple phones, although a few early smartphones were around such as the Nokia Communicators. Some could now access the internet via the newly introduced Wireless Application Protocol, but the slowness over the old phone networks, reduced experience compared to a normal web browser, and the limitation of having to use special cut-down sites meant it was a patchy success at best.

By 2000 the internet had left the preserve of the more technically minded and was becoming mainstream. Access was still mainly via dial up connections, but ISDN and cable modems were becoming more popular. Microsoft was close to winning the so-called Browser Wars with Internet Explorer 5. The dot.com bubble was at its peak as people scrabbled for a foothold in this rapidly growing new medium, but would burst only a few months later. Few people had much idea of what would actually work on the internet. Much of the investment capital was thrown at all sorts of ideas, in the hope that some of the companies would survive and go on to dominate. Inevitably, many of the companies folded quickly after the money had run out, for example the notorious Boo.com who spent $188 million in six months. In contrast Amazon, then around 5 years old, was criticised for only having slow and steady growth instead of spending as much as possible to get market share. Google was a fairly new search facility – only 40 employees – but becoming popular as its new page ranking technology allowed people to find the most relevant information. The internet had also started to make an impact in mainstream life, for example the early Blog, the Drudge Report, had broken the Lewinsky scandal a couple of years earlier. Many news sources were setting up a web-presence, in the UK notably The Guardian and the BBC. Wikipedia was still a year off, although the underlying wiki technology had been around for a while, after being invented in order to aid collaborative Pattern writing.

Due to the internet, computer security was becoming much more important with the spread of viruses and trojans made much easier by the improved interconnectedness, enhanced by worries that due to the dominance of Microsoft operating systems and application hosts for scripts, a mono-culture effect could mean a major outbreak could spread quickly and widely. Governments too were concerned with security, but mainly so they could intercept and read people's telephone calls and emails. Strong encryption algorithms were even classified as munitions and covered by arms trading legislation.

Right here, right now

That was then, where are we now? Moore's law has continued to hold, and the number of transistors on a modern Intel chip now number around 750 million! Significantly though, the clock speed hasn't maintained the rapid rise of a decade ago – after peaking at a bit over 3GHz, the speeds have dropped back to around 2.9MHz as heat dissipation became a major problem. So instead of relying on clock speed increases to improve performance, chip manufacturers have had to use increasingly

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

complicated techniques, such as instruction lookahead and speculative execution, hugely expanded on-chip memory caches – we're now talking multiple MB – to avoid having to wait for the main memory, and having multiple cores to allow true multitasking. This is where all those extra transistors have gone – instead of implementing a much bigger instruction set, most are actually memory and copies of the main processing cores, and also the tricky algorithms to improve instruction and memory throughput. In contrast the instruction sets haven't expanded as much, although they have gained some extra multimedia-oriented facilities and processing large data sets.

As well as the sort of complex chips in PCs, much smaller, simpler and more focused chips are now much more common, whether it's an ARM RISC chip in a mobile phone, a custom ASIC for implementing bluetooth or GPS. As well as simple chips, it's easier to combine off-the-shelf modules on a single piece of silicon to create a so called System On A Chip. These have allowed a wide range of powerful, yet cheap and small consumer devices to be released.

The amount of main memory has also expanded so many PCs now come with 4GB, which is the limit of addressable space for 32bit pointers. Disk sizes have shot up faster than Moore's Law, helped by applying exotic techniques such as spintronics, with 1TB disks being easily available, and even laptops come with 256GB disks. A major problem with disks and storage is now the transfer rate.

The actual form factors have changed a lot though. As well a the traditional tower and big laptop, there are now tiny form factors such as the Shuttle boxes, Mac Mini and Acer Revo, and all-in-one computer and monitors have become more widespread. Laptops have become smaller, lighter, and more powerful, with a new niche of cheap 'Net-books'. Having multiple monitors has become much more common, initially via graphics cards with multiple outputs and then software solutions such as DisplayLink. USB 2 itself has became ubiquitous and is used to connect a wide range of consumer electronics as well as PC peripherals. Version 3, theoretically ten times faster, has been defined and new products are starting to be rolled out.

In terms of operating systems, Windows 7 is now out to replace Vista, although XP remains popular on lower powered machines, OSX has gone through several iterations, and there are others such as Ubuntu's version of Linux, the iPhone operating system, and Google's Android and upcoming ChromeOS.

In terms of raw numbers, the iPhone is not that big a player in the massively expanded mobile phone market, but very important in terms of influence. Smartphones had been around for ages, but the iPhone made the leap to making them usable and desirable with its large touch sensitive screen, smooth graphics and UI, and excellent design. Plus with 3G and WiFi access from it and other phones, mobile internet access is now easy to do, and with other technologies such as built in camera, GPS and compass, portable 'information appliances' are now a reality. By the time you read this, Apple will have launched its tablet computer – could this change the new eReader segment and portable computer market in a similar way?

Google has been one of the biggest successes of the decade. The range of services it now provides is stunning, although there are increasing worries about data security in the cloud, and how much personal data it keeps with important privacy concerns.

Which shows how important the internet now is in people's everyday lives. With the advent of common ASDL and cable broadband, plus upgrades to mobile phone networks, and wired and wirelss networks in the home and wireless hotspots in public places, virtually all PCs and mobile devices are now connected to a vastly expanded range of information and services. Whether it's a developer looking up some up-to-date documentation, doing your tax return online, getting medical advice via NHS direct and booking a doctors appointment, uploading a video of an anti-government protest, or a Twitter during the Haiti earthquake, someone checking IMDB to cheat in a pub quiz, or looking up the nearest restaurant and checking reviews, the internet is now an essential service. Which does have its dangers – cyber attacks and information theft are much more common and more severe than ever before, although these are mitigated to some extent by the advance of the defences in operating systems, firewalls, and anti-malware. It is a continuing arms race, though.

If I had to name the one disruptive change over the last decade, it must surely be the roll out of fast, pervasive networks, leading to permanent connection to a vast information source, and other people. Virtually all the other big changes are built upon this.

A look into the future

Predicting disruptive technology and future directions is very hard. Sometimes it's because it hasn't been thought of yet. Sometimes it exists but needs other things to happen for them to become important. The last big one was probably the internet, which took 40 odd years. Something in the medical or biotech arenas might be next – the price of genome sequencing is dropping fast, opening up lots of possible developments. Integrating GPS and cameras into powerful phones is opening up some interesting syntheses – think of William Gibson's *Virtual Light*. RFID has been promising much and gaining footholds in niches. But would it change everyone's life significantly? Parallel and distributed computing will be important, whether it's multi core, grid or cloud computing. But what will be the killer application? If I knew that, I'd be rich soon!

But you want a prediction. I think the major technological changes for the next decade or so will be driven by nanotech. Improvements in things such as battery efficiencies, LCD screens, chip development, disk densities, photovoltaic efficiencies, etc, have already been happening due to its application, but I expect it will accelerate and affect many things, and result in the next step change in small, efficient, ubiquitous computing.

One Approach to Using Hardware Registers in C++

Testing increases software reliability. Martin Moene presents a technique for checking the control of hardware.

Software that accesses hardware registers is not always written as clearly as one would like. A cause for this may be the assumption that using an abstraction for the register degrades performance too much. Also such code often lacks good support for testing, which is aggravated by the write-only property of many registers that complicates verifying if the software operates correctly. This article presents the approach that we use to addresses these issues in our software for scanning probe microscopy.

Scanning Probe Microscopy

To form an idea of a scanning probe microscope [SPM], you may recall the old stereo vinyl-record player with its needle picking up small height variations of the record groove. Now imagine the needle out the groove, making a scanning movement in two directions over a part of the record's surface, somewhat like the head of an inkjet printer scanning over a sheet of paper. This resembles the scanning probe that scans a sample and probes its height variations with a tip. Shrink it to the atomic scale of nanometres and it's called a microscope. Figure 1 illustrates the scanning movements.

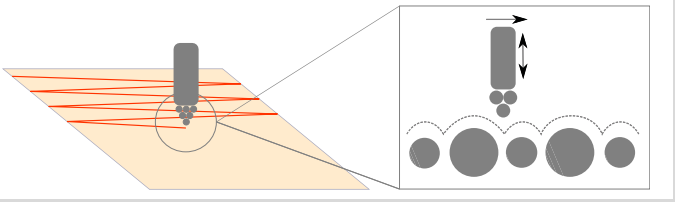

Figure 1

The seeing actually is more like feeling and one of the sensing methods uses the tunnel current [TC] that flows between tip and sample for that purpose. This is called scanning tunneling microscopy (STM). Another type of scanning probe microscopy is atomic force microscopy (AFM), where the sensing method relies on interaction forces between tip and sample if they are close. Before scanning a material's surface, the tiny tip is brought towards the area of interest in a delicate process that's called *approach*.

The strength of the measured interaction between tip and surface is plotted against the scan coordinates in an intensity graph that then represents a view of the surface's topography.

Further, the Leiden Interface Physics [IP] group has designed and built custom electronics to perform scanning probe microscopy measurements as fast as possible. Material with a very even surface allows the recording of movies with up to 80 images of 128×128 pixels squared per second, while still obtaining atomic resolution. With these and several other techniques in place, changes on the surface of a material can be 'videoed'while for example different kinds of gas are flowed over the surface in succession, leading to new discoveries [Frenken05].

Martin Moene has been programming professionally since 1983, mostly in C++. He has a background in electronics engineering, and most programming revolves around instrucment control, image processing, crunching numbers and sometimes administrivia. Martin can be contacted at m.j.moene@eld.physics.LeidenUniv.nl

In control

The video-rate SPM controller used for these measurements consists of a rack with several modules or cards. There are two computer buses in the rack, one called STM-bus and the other ADC-bus. STM-bus is a bit of a misnomer, as the controller is not limited to STM measurements. The cards connected to the STM-bus are used to generate the signals to perform the tip (sample) scanning movement and the timing of the measurements. The ADC-bus handles the signals measured by several Analog to Digital Converters (ADCs) on that bus. While scanning, the measured values are transfered to the computer under DMA[1].

Each bus has a bus controller card that is connected via a pair of glass fibers to a PCI interface card in the computer that controls the measurement (Figure 2). A spin-off called Leiden Probe Microscopy [LPM] now produces and markets the SPM controller.

At the same time and as part of the project, C++ software has been developed to interface with the electronics and to perform and analyse STM and AFM measurements.

Control fades

As research objectives change, and new insights in research methods arise, the software has to change. However the software is showing its age and it is becoming increasingly difficult to adapt: it incurred technical debt [Fowler09]. In its current form, the software is also not very well suited to use the SPM controller for measurements outside the field of scanning probe microscopy. Eventually we decided to redevelop the parts of the software that interface with the electronics and that provide the basic scanning probe microscopy functionality.

Regaining weight

As the existing software has no supporting unit tests – [Feathers04] calls this legacy software – and the gap between it and the desired situation was quite large, we decided not to refactor [Fowler99] the existing code but instead completely redevelop it. We also took the chance to move from Microsoft's Visual C++ version 6 to version 8, which later can be replaced more easily with an even newer version. Where it supports our needs well, we decided to prefer to use [Boost] libraries over other possible libraries. This guideline helped us to choose Boost.Test over for example [GTest] for unit testing.

In the remainder of the article we'll look at our approach to development and testing of the software that controls the electronics with an emphasis on accessing hardware registers and testing the bits and bytes that eventually will flow through them.

1.Direct Memory Access, access system memory independent of the CPU.

A hardware register is a kind of memory element, although its implementation may differ from memory used for temporary storage in a computer

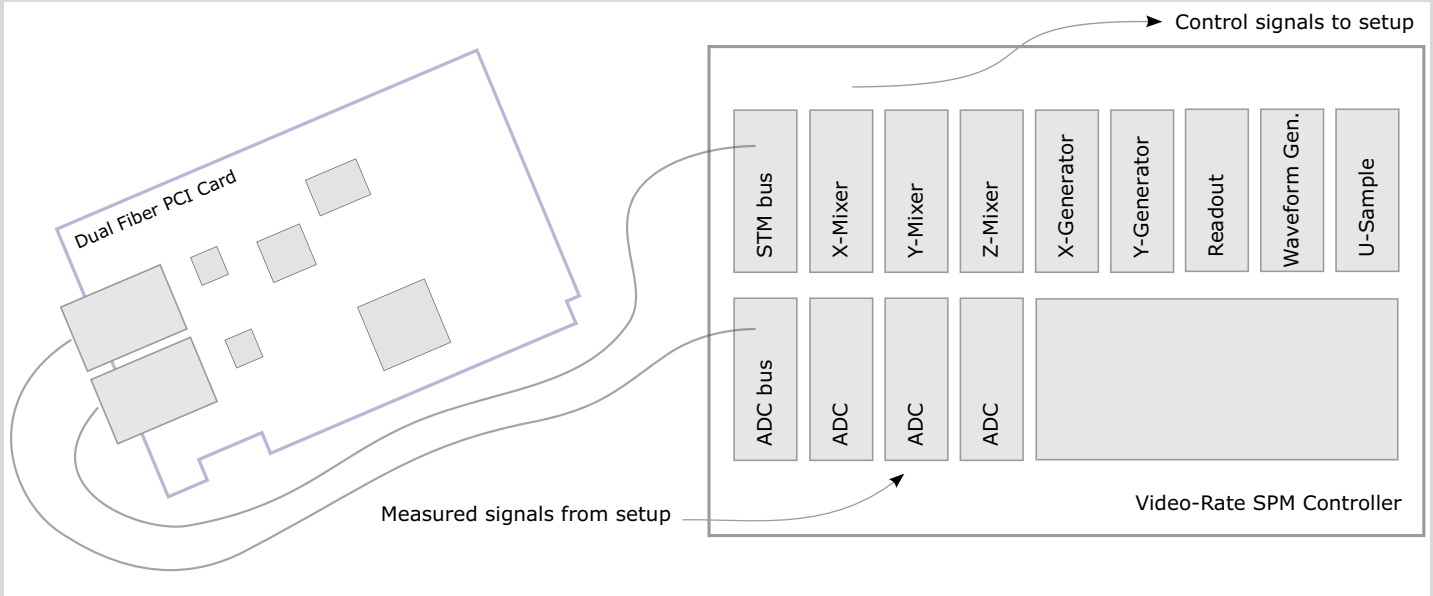

Figure 2

While writing this article I encountered 'A Technique for Register Access in C++', by Pete Goodliffe, which has the following introductory sentence: 'This article discusses a C++ scheme for accessing hardware registers in an optimal way' [Goodliffe05]. It contains a nice introduction to hardware registers and how they are accessed. Its emphasis on the limitations of embedded software is not a concern here.

Register diversity

A hardware register is a kind of memory element, although its implementation may differ from memory used for temporary storage in a computer [HR]. The key properties of a register in this discussion are its data and address width and the method used to access it. Common register access methods are memory-mapped I/O, port-mapped I/O and bus-separated access. Further it is quite common that information written to a register cannot be read-back from it, complicating read/modify/write operations and testing [Roberts]. And for most registers there isn't an easy way to electronically check the result of how we configure it.

A rough analysis was done of the functions of the various cards involved and of their hardware registers' properties. We counted 34 registers, of which 13 use simple word-wide access. The remaining 21 registers are control and status registers and have multiple functions. As much as possible, we would like to be able to use non-related functions within a single register separately from each other. On the other hand sometimes the microscope or domain behaviour requires that what may be disparate register functions or even disparate functions in separate registers, must change in concert.

There are memory-mapped registers on the PCI interface card and registers on the cards in the SPM rack that are accessed via the PCI interface card. The registers on the PCI card are 32-bit wide and have a 32-bit address, whereas the registers on the SPM cards have a 16-bit data type and an 8-bit address. It would be nice if we can use the same register abstraction for the registers on the PCI interface card and the registers on the cards that are accessed via that PCI interface card.

In our search for an approach that works, these are the guiding ideas:[1]

- **Use**: provide a register abstraction with useful (bitwise) operations such as bittest, bitset and masked variants thereof and use it for all register types.
- **Test**: make register access testable and actually test it via automated unit tests.
- **Access policy**: separate the actual register access method from the register abstraction.
- **Performance**: ensure register access speed that is comparable to pointer-based register access for memory-mapped registers.

Show, don't (just) tell

What the test environment should provide is simple. Initially, while we develop the code, the test environment is allowed to relax [TeX] and to

1.I almost wrote the words 'flexible' and 'reuse' here, however: 'The word flexible is like reuse: it should alert you that something nebulous is probably up. Classes and functions are not designed to be flexible, they are designed for a purpose: flexibility is not a purpose, nor is it either a quality or a quantity; it is a bucket term, a catch all, snake oil.' See [Henney02].

```
prompt>test --log_level=message --run_test=*/*Stm*/*Read*
Running 1 test case...
Inspect> testThatReadUsesRightRegisterAccessSequence:
Inspect> 1: 0x0044 <-- 0x0001 0b0000000000000001 1 | clear fifo data
Inspect> 2: 0x0044 <-- 0x0000 0b0000000000000000 0 | is available flag
Inspect> 3: 0x0044 --> 0x0002 0b0000000000000010 2 | wait for write
Inspect> 4: 0x0044 --> 0x0000 0b0000000000000000 0 | fifo not full
Inspect> 5: 0x0060 <-- 0x82aa0000 0b10000010101010100000000000000000 2192179200 | write read command
Inspect> 6: 0x0044 --> 0x0002 0b0000000000000010 2 | wait for address
Inspect> 7: 0x0044 --> 0x0002 0b0000000000000010 2 | to become
Inspect> 8: 0x0044 --> 0x0000 0b0000000000000000 0 | accepted
Inspect> 9: 0x0044 --> 0x0000 0b0000000000000000 0 | wait for data to
Inspect> 10: 0x0044 --> 0x0001 0b0000000000000001 1 | become available
Inspect> 11: 0x0060 --> 0x3355 0b0011001101010101 13141 | note host to fiber byte swap
*** No errors detected
```

Listing 1

```
prompt>test
Running 3 test cases...

Test-fail.cpp(52): error in "testThatRegisterAssignmentWritesProperValueToRegister": check {
spy.getAllExpectations().begin(), spy.getAllExpectations().end() } == {
spy.getAllOccurrences().begin(), spy.getAllOccurrences().end() } failed.

Mismatch in a position 0: [write,0x17,0x123] != [write,0x17,0x246]

*** 1 failure detected in test suite "Master Test Suite"
```

Listing 2

just report the register access that it sees. Thus we can compare register access with the manual description and existing code, reason about it and easily spot and understand any errors we make. Gaining trust in what we wrote, we add register read and write expectations that specify the programmed behaviour and ensure that it is tested (not relaxing anymore). With development completed, visual feedback on the register operations is turned off and only a failing test case may draw our attention.

Although the way of working resembles test driven development [TDD], I merely regard it as visual inspection driven implementation (VIDI) or to overload the term, probe driven development (PDD). Seeing what actually happens to the register first helps to implement the required behaviour correctly.

Listing 1 presents some successfully verified test output that shows the register interaction for a call to the function read(address) of the PCI interface card in the computer to obtain a value from a card in the SPM controller.

At the left of the arrow is the address to read from (`-->`) or write to (`<--`), at its right is the register content in hexadecimal, binary and decimal notation. At item 5, in `0x82aa0000`, `0x82` is the STM-bus read command, `0xaa` the card address to read from and `0x5533` at item 11 is the value obtained from the card by the simulated read. At the far right, remarks explain what should be happening. Note the explicative name of the test: `testThatReadUsesRightRegisterAccessSequence` [Henney09].

Listing 2 shows a failing test where the value written to the register (`0x246`) is different from the expected one (`0x123`).

Register class declaration

Ah, finally we get to see some bits of code! Listing 3 shows the `Register` class declaration.

A `Register` object is defined by its data type (D), address type (A) and the concept (CP) that defines how the register is accessed, e.g. as memory or in another way.

To test registers access, the actual reading from and writing to the hardware registers must be intercepted. The user-provided access policy class make this possible [Henney06, Henney08]. Another approach would be to make the register access an abstract interface. However, the policy approach potentially provides better performance as the compiler can optimise away function calls, whereas it may not do that with the virtual function calls of an abstract interface.

As a register object represents a single hardware register, we prevent copying by inheriting the register class from `boost::noncopyable`.

Channel concept

How exactly registers are accessed is governed by the channel concept. It prescribes that the policy class provides a `read()` method and a `write()` method with proper data and address types (Listing 4).

That's practically all there is to it, the type of the policy class itself is not relevant. This kind of polymorphism – compile-time polymorphism in case of C++ – is called duck typing [DT]: if it walks, quacks and swims like a duck, it's probably a duck.

Memory-mapped register access

Some hardware registers are accessed in the same way as memory. These memory-mapped registers need only a simple access policy, such as the one shown in Listing 5.

Here, the address type is derived from the data type. With `volatile` in the intermediate `RegisterType` declaration we inform the compiler that

```
/**
 * register type.
*/
template
< typename D // data type
, typename A = D* // address type
, typename CP = MemoryChannel<D,A>// channel policy
>
class Register;
```

Listing 3

```
/**
 * the duck in the channel.
 */
struct ChannelConcept
{
  typedef sometype DataType;
  typedef sometype AddressType;
  DataType read ( AddressType );
  void write( AddressType, DataType );
};
```

Listing 4

the value in the memory location may change without the compiler being aware of it. The effect is that the compiler will not optimise away any reads from the location that it may consider redundant.

As we'll see shortly, the **Register** class can use channel policy class objects that are created either internally or externally. Some policy classes have no need for object member data and can use static member functions. The **MemoryChannel** class is an example of this: there's no need for an externally initialized object of it.

Intermezzo: Template type parameter or template template parameter

Early in development, I made the channel policy a template template parameter, so that the **register** class governs the channel's data and address types. For memory-mapped registers this seems a natural choice. However, is it also a good choice for a channel that has its own fixed data and address types?

Then when I wanted to use a spied-upon fiber interface card class as the channel policy of a card class, I was in trouble, because we are no longer feeding the card's class a template class but a type instead. It was time to consult the ACCU-general mailing list and ask for reasons and consequences of choosing between a template type parameter and template template parameter. James Dennett's answer exactly mentioned what I was experiencing: a template type parameter gives extensibility/flexibility, or put otherwise the choice for a template template parameter results in non-extensibility and inflexibility [Dennett09]. Thus the channel policy template parameter became a type.

Now that data and address types for the register and the channel can be specified separately, this could introduce a conversion. However, for simplicity it is just checked at compile-time if the types are equivalent using for example **BOOST_STATIC_ASSERT(boost::is_same<D, typename CP::D>::value);**

```
/**
 * transport for memory mapped registers.
 */
template
< typename D // data type
, typename   // A address type unused
>
class MemoryChannel
{
public:
  typedef D DataType;
  typedef volatile DataType RegisterType;
  typedef RegisterType* AddressType;
  static DataType read( AddressType address )
  {
    return *address;
  }
  static void write( AddressType address,
     DataType data )
  {
    *address = data;
  }
};
```

Listing 5

Register class implementation

Listing 6 presents the implementation of the **Register** class. There are a couple of things to note.

- there are two constructors: one without channel object, one with channel object parameter
- the destructor contains a call to method **checkTypes** that statically asserts that data and address type of register and channel policy match.
- the class normally caches the value written to the hardware register to compensate for the fact that the registers are write-only; if a registers can also be read it usually has a different meaning, e.g. it is a status value instead of the written control value.

And of course, there are the methods to read and write the register as a whole, or test, set and clear bits, or groups of bits.

Although a register just has a single address, the constructor takes both an address and an offset. The rationale behind it is to concentrate address computations at a single point in the register class and not spread it over the classes that use the register class.

```
template
< typename D
, typename A = volatile D*
, typename CP = MemoryChannel<D,A>
>
class Register : public boost::noncopyable
{
public:
  typedef D DataType;
  typedef A AddressType;
  typedef CP ChannelType;
  Register( AddressType address, int offset,
     DataType data = 0 )
  : m_channel_smartptr()
  // Note: must be declared before m_channel
  , m_channel( createChannel() )
  , m_address( computeAddress( address,
     offset ) )
  , m_cache( data )
  {
  }
  Register( ChannelType& channel,
     AddressType address, int offset,
     DataType data = 0 )
  : m_channel_smartptr()
  , m_channel( channel )
  , m_address( computeAddress( address,
     offset ) )
  , m_cache ( data )
  {
  }
  ~Register() {
    checkTypes();
  }
  static void checkTypes() {
    BOOST_STATIC_ASSERT( ( boost::is_same<D,
       typename CP::DataType>::value ) );
    BOOST_STATIC_ASSERT( ( boost::is_same<A,
       typename CP::AddressType>::value ) );
  }
  operator DataType() {
    return read();
  }
```

Listing 6

```
  Register& operator= ( DataType data ) {
    write( data );
    return *this;
  }
  AddressType address() const {
    return m_address;
  }
  DataType cache() const {
    return m_cache;
  }
  DataType read() {
    return m_channel.read( m_address );
  }
  void write() {
    m_channel.write( m_address, m_cache );
  }
  void write( DataType value ) {
    m_channel.write( m_address,
       m_cache = value );
  }
 void write_nc( DataType value ) {
    m_channel.write( m_address, value );
  }
  bool bittest( int bit ) {
    return 0 != ( read() & bitmask( bit ) );
  }
  void bitclear( int bit ) {
    write( m_cache & ~bitmask( bit ) );
  }
  void bitset( int bit ) {
    write( m_cache | bitmask( bit ) );
  }
  bool masktest( DataType mask ) {
    return mask == ( read() & mask );
  }
  void maskclear( DataType mask ) {
    write( m_cache & ~mask );
  }
  void maskset( DataType mask ) {
    write( m_cache | mask );
  }
  void maskset( DataType clearmask,
     DataType setmask ) {
    m_cache &= ~clearmask;
    maskset( setmask );
  }
  static DataType bitmask( int bit ) {
    return 1 << bit;
  }
  static AddressType computeAddress(
     AddressType base, int offset ) {
    return base + offset;
  }

private:
  ChannelType& createChannel() {
    m_channel_smartptr.reset(
       new ChannelType() );
    return *m_channel_smartptr;
  }

private:
  // to be replaced by std::unique_ptr:
  std::auto_ptr<ChannelType> m_channel_smartptr;
  ChannelType& m_channel;
  AddressType m_address;
  DataType m_cache;
};
```

Listing 6 (cont'd)

```
template < typename CP >
class DualFiberLinkImpl
{
  typedef CP ChannelType;
  typedef Register< pci_data_t, pci_address_t,
     CP > RegisterType;
  class BusStatusRegister : private RegisterType
  {
  private:
    enum EStatusRegister
    {
      eBit_DataIsAvailable = 0,
      eBit_WriteFifoIsFull = 1,
      eBit_ErrorHasOccurred = 2,
      eBit_ClearDataIsAvailable = 0,
      eBit_ClearErrorHasOccurred = 2,
      ...
    };
  public:
    BusStatusRegister( ChannelType& channel,
       pci_address_t address, int offset )
    : RegisterType( channel, address,
       offset ) {;}
    bool fiberDataIsAvailable() const {
      return bittest( eBit_DataIsAvailable );
    }
    ...
    void clearFiberDataIsAvailable() {
      bitset ( eBit_ClearDataIsAvailable );
      bitclear( eBit_ClearDataIsAvailable );
    }
  };
  ...
  enum ERegisterOffset
  {
    eRegOff_BusStatus = 1,
  };
public:
  DualFiberLinkImpl( ChannelType& channel,
     const pci_address_t address )
  : m_regBusStatus ( channel, address,
     eRegOff_BusStatus )
  ...
  {
  }
  bool fiberDataIsAvailable() const
  {
    return m_regBusStatus.fiberDataIsAvailable();
  }
  void clearFiberDataIsAvailable()
  {
    m_regBusStatus.clearFiberDataIsAvailable();
  }
private:
  BusStatusRegister m_regBusStatus;
  ...
};
```

Listing 7

All in all, not too exciting a class. It's the combination of register operations, registers access and its testing that makes it interesting.

Using class Register

Now where do all these preparations bring us to? Listing 7 presents a small part of class **`DualFiberLinkImpl`** for the computer interface PCI card that connects the computer to the video-rate SPM controller.

In normal use, the class will be instantiated with the MemoryChannel policy class to provide access to the memory-mapped PCI registers.

Configurable register spy

The channel concept not only allows for different ways to access real registers, it also provides the means to bring the register access into our test framework. A register spy class is a channel policy and besides that it can contain and provide the expected register access operations as well as the actually occurred register access operations [Meszaros07]. The macro call **`SPM_CHECK_REGISTER_SPY_EXPECTATIONS(spy)`** checks if the programmer-defined expectations are met (Listing 8).

Boost.Test

As already mentioned, we're using Boost.Test as the test framework. It nicely supports the described way of working through its message and test macros and its command line options. While developing, we use option **`--log_level=message`**, later on when used as regression test we use **`--log_level=error`**, which is the default. With option **`--run_test=spec`** we can select one or more tests to run instead of running all tests. For example, option **`--run_test=*/*Stm*/*Read*`** selects the tests with **`Read`** in their name from the (sub) test suites that have **`Stm`** in their name.

Sometimes the tests specified initially were wrong and failed, whereas the code to test was correct. I don't think this is a bad thing, it just makes you all the more conscious of what the code does. One thing I was able to spot immediately occurred when we moved from a Windows-API-based lock to the Boost-based lock. The read operation for the fiber interface PCI card halted where it previously had no problem. It appeared that I had inadvertently chosen a non-re-entrant mutex from Boost.Thread, whereas the **`CRITICAL_SECTION`** previously used for the mutex is re-entrant.

A test example

The Boost.Test main program (`Test-main.cpp`) is joyfully simple. The actual groups of related tests (test suites) are located in separate source files, such as `Test-simple.cpp` in Listing 9. Compile and link the source files with `Test-main.cpp` as the first to obtain the test program with all test suites.

```
/**
 * the spy we love.
 */
template < typename D, typename A >
class RegisterSpy;

// main spy operations:
void relax( bool relax = true );
void addReadExpectation ( AddressType address,
   DataType data, std::string remark = "" );
void addWriteExpectation( AddressType address,
   DataType data, std::string remark = "" );

// macros:

// the name of the current test case.
#define SPM_TEST_CASE_NAME ...

// issue test message; streams its argument.
#define SPM_TEST_MESSAGE( arg ) \
   BOOST_TEST_MESSAGE( "" << arg )

// issue test message:
// "'prefix' testcase_name" << 'postfix'.
#define SPM_TESTCASE_MESSAGE( prefix, postfix ) \
   SPM_TEST_MESSAGE( prefix << \
   SPM_TEST_CASE_NAME << postfix )

// match expectations and occurrences.
#define SPM_CHECK_REGISTER_SPY_EXPECTATIONS(
   spy ) ...
```

Listing 8

In its most basic usage **`spy.relax()`** is called before the register is used and the register spy just records the access to one or more registers. However, here the spy is provided with a sequence of expectations of addresses and values that should be written and read and at the end of the test these expectations are matched with the actual register accesses to report any discrepancy (Listing 9).

Space and time efficiency

The design of the **`Register`** class assumes that calls to the channel policy class are optimised away by the compiler. This results in code that's both smaller and faster because of the absence of a function call. Processor cache size limits also reward smaller code with faster execution.

Space efficiency

Each object of the **`Register`** class contains a smart pointer used for internally created channel policy objects, a reference to the channel, the register's address and the cache for the value written to the register. Would another approach, for example one where less information is stored in the register objects, lead to overall smaller code? I don't know. I didn't look into it because size per se is not a big concern to me. The chosen approach

```
// File: Test-main.cpp
#define BOOST_TEST_MAIN Master Test Suite
#include <boost/test/unit_test.hpp>
// File: Test-minimal.cpp
#include <iostream>      // std::cout
#include "Register.h"    // class Register
#include "RegisterSpy.h" // class RegisterSpy
#include "Test-common.h"
                    // SPM_TEST_INSPECT_MESSAGE
#include <boost/test/unit_test.hpp> // Boost.Test
typedef int DataType;
typedef int AddressType;
typedef spm::tdd::RegisterSpy< DataType,
   AddressType > RegisterSpyType;
typedef spm::Register < DataType, AddressType,
   RegisterSpyType > RegisterType;
const AddressType base ( 0x10 );
const int offset ( 0x07 );
const DataType initial( 0xe5 );
const AddressType address(
   RegisterType::computeAddress( base,
   offset ) );
BOOST_AUTO_TEST_SUITE( Register )
BOOST_AUTO_TEST_SUITE( Minimal )
struct Fixture
{
  Fixture() : spy( "Inspect>" ) , reg( spy, base,
     offset, initial ){;}
  ~Fixture() { SPM_TEST_MESSAGE( spy ); }
  RegisterSpyType spy;
  RegisterType reg;
};
BOOST_FIXTURE_TEST_CASE(
testThatRegisterAssignmentWritesCorrectValueToReg
ister, Fixture )
{
  SPM_TESTCASE_MESSAGE(
     "Inspect> ", " (Pass):" );
  const DataType value( 0x123 );
  spy.addWriteExpectation( address, value,
     "assign value to register" );
  reg = value;
  SPM_CHECK_REGISTER_SPY_EXPECTATIONS( spy );
}
BOOST_AUTO_TEST_SUITE_END() // Minimal
BOOST_AUTO_TEST_SUITE_END() // Register
```

Listing 9

leads to quite simple code for register manipulation, as much is abstracted into functions to build on, so my impression is that it is size-efficient.

Register performance

Register access is at the heart of many operations, so performance may be an issue. Moving a slider that controls a voltage in the setup to pan (or zoom, rotate etc) the scanned area should be a smooth operation. Say run-time performance and the very next word is: measure. I timed the various operations the register class provides as well as comparable pointer-based memory access statements. Table 1 lists these measurements. Note that the tests access conventional memory as opposed to registers on a 66 MHz PCI bus (15 ns period). However our prime interest is to compare the performance of the `Register` approach with the presumed efficient way it is done in the legacy code.

Note that the timing also depends on other tasks running on the computer and therefore the test program was run 100 times to spread out the measurements in time. The register class performs quite well compared to the pointer-based access.

Table 2 presents the resulting assembly code. It appears that often the code generated for the `Register` class is the same or almost the same as for the equivalent pointer based statements.

Conclusion

Using fairly main-stream C++ constructs we provide a hardware register abstraction that enables us to write classes that represent hardware with a clear and regular design. The register abstraction allows for different methods to access the registers and with this it also provides the means to test register read and write access. And thanks to compiler optimisation, for memory-mapped registers access it has a performance akin to pointer-based access. So if we can speak of any degradation of performance, it is offset by enhanced testability. ■

Operation memory access times on computer running Windows-XP SP3 with a 2.3 GHz AMDAthlon(tm) 64 X2 Dual Core Processor 4400+ and 2 GByte RAM. The program was compiled with MSVisual C++ 8, with options -O2 -EHs. Times are in [ns].

#	Median [ns]	Operation
1	1.31	*p = x
2	1.31	reg.write_nc(x)
3	1.31	*p = cache = x
4	1.595	reg.write(x)
5	1.6	reg = x
6	1.36	cache = *p
7	1.75	cache = reg.read()
8	1.75	cache = reg
9	1.36	b = 0 != (*p & 0x01)
10	1.57	b = reg.bittest(0)
11	1.75	*p = (cache | 0x02)
12	1.75	reg.write_nc(reg.cache() | 0x02)
13	2.29	*p = cache |= 0x04
14	2.41	reg.bitset(3)
15	2.31	*p = cache |= 0x0f
16	2.41	reg.maskset(0x0f)
17	2.9	*p = cache = ((cache & ~0xf0) | 0x30
18	2.98	reg.masksset(0xf0, 0x30)

Table 1

Acknowledgements

Thanks to editor Ric Parkin for the gentle guidance of a first-time Overload author. Also thanks to Gert Jan van Baarle and Joost Frenken for their

Assembly code when compiled with VC8, options -O2 -EHa.

1	*p=x	
	mov	ecx, DWORD PTR [esi+12]
	mov	DWORD PTR [ecx], eax
2	reg.write_nc(x)	
	mov	ecx, DWORD PTR [esi+28]
	mov	DWORD PTR [ecx], eax
3	*p = cache = x	
	mov	ecx, DWORD PTR [esi+12]
	mov	DWORD PTR [esi+4], eax
	mov	DWORD PTR [ecx], eax
4	reg.write(x)	
	mov	ecx, DWORD PTR [esi+28]
	mov	DWORD PTR [esi+32], eax
	mov	DWORD PTR [ecx], eax
5	reg = x	
	mov	ecx, DWORD PTR [esi+28]
	mov	DWORD PTR [esi+32], eax
	mov	DWORD PTR [ecx], eax
6	cache = *p	
	mov	ecx, DWORD PTR [esi+12]
	mov	edx, DWORD PTR [ecx]
	mov	ecx, DWORD PTR [esi+12]
	mov	DWORD PTR [esi+4], edx
7	cache = reg.read()	
	mov	eax, DWORD PTR [esi+28]
	mov	eax, DWORD PTR [eax]
	mov	DWORD PTR [esi+4], eax
8	cache = reg	
	mov	eax, DWORD PTR [esi+28]
	mov	eax, DWORD PTR [eax]
	mov	DWORD PTR [esi+4], eax
9	b = 0 != (*p & 0x01)	
	mov	ecx, DWORD PTR [esi+12]
	mov	edx, DWORD PTR [ecx]
	mov	ecx, DWORD PTR [esi+12]
	and	dl, 1
	mov	BYTE PTR [esi], dl
10	b = reg.bittest(0)	
	mov	eax, DWORD PTR [esi+28]
	mov	eax, DWORD PTR [eax]
	and	eax, 1
	mov	BYTE PTR [esi], al
11	*p = (cache | 0x02)	
	mov	ecx, DWORD PTR [esi+4]
	mov	edx, DWORD PTR [esi+12]
	or	ecx, 2
	mov	DWORD PTR [edx], ecx

Table 2

support in writing this article and their rapid review that made it possible to publish it one issue earlier than first envisioned.

Source code

The article's source code is available as a `tar.gz` file from the following web page: http://www.eld.physics.LeidenUniv.nl/~moene/accu/overload/95/register/

12	reg.write_nc(reg.cache() | 0x02)	
	mov	eax, DWORD PTR [esi+32]
	mov	ecx, DWORD PTR [esi+28]
	or	eax, 2
	mov	DWORD PTR [ecx], eax
13	*p = cache |= 0x04	
	mov	ebx, 4
	or	DWORD PTR [esi+4], ebx
	mov	eax, DWORD PTR [esi+4]
	mov	edx, DWORD PTR [esi+12]
	mov	DWORD PTR [edx], eax
14	reg.bitset(3)	
	mov	eax, DWORD PTR [esi+32]
	mov	ecx, DWORD PTR [esi+28]
	or	eax, 8
	mov	DWORD PTR [esi+32], eax
	mov	DWORD PTR [ecx], eax
15	*p = cache |= 0x0f	
	mov	edi, 15 ; 0000000fH
	or	DWORD PTR [esi+4], edi
	mov	eax, DWORD PTR [esi+4]
	mov	edx, DWORD PTR [esi+12]
	mov	DWORD PTR [edx], eax
16	reg.maskset(0x0f)	
	mov	eax, DWORD PTR [esi+32]
	mov	ecx, DWORD PTR [esi+28]
	or	eax, 15 ; 0000000fH
	mov	DWORD PTR [esi+32], eax
	mov	DWORD PTR [ecx], eax
17	*p = cache = ((cache & ~0xf0) | 0x30)	
	mov	eax, DWORD PTR [esi+4]
	and	eax, -193 ; ffffff3fH
	or	eax, 48 ; 00000030H
	mov	DWORD PTR [esi+4], eax
	mov	edx, DWORD PTR [esi+12]
	mov	DWORD PTR [edx], eax
18	reg.maskset(0xf0, 0x30)	
	mov	eax, DWORD PTR [esi+32]
	mov	ecx, DWORD PTR [esi+28]
	and	eax, -193 ; ffffff3fH
	or	eax, 48 ; 00000030H
	mov	DWORD PTR [esi+32], eax
	mov	DWORD PTR [ecx], eax

Table 2 (cont'd)

References and further reading

[Boost] Boost free peer-reviewed portable C++ source libraries, http://www.boost.org/.

[Dennett09] James Dennet, accu-general mailing list, December 2009, http://lists.accu.org/mailman /private/accu-general/2009-December/018308.html.

[DT] Duck Typing (Wikipedia), http://en.wikipedia.org/wiki/Duck_typing.

[Feathers04] Michael Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2004.

[Fowler99] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison–Wesley Professional, 1st edition, 1999.

[Fowler09] Martin Fowler, TechnicalDebtQuadrant, October 2009, http://martinfowler.com/bliki /TechnicalDebtQuadrant.html.

[Frenken05] Joost Frenken et al., Pushing the limits of SPM, *Materials Today*, May 2005, http://www.physics.leidenuniv.nl/sections/cm/ip/group/PDF/Materials%20Today/%282005%2920.PDF; For a more in-depth article, see [Rost09].

[Goodliffe05] Pete Goodliffe. 'A Technique for Register Access in C++', ACCU *Overload 68*, August 2005, http://accu.org/index.php/journals/281.

[GTest] 'Google C++ Testing Framework', http://code.google.com/p/googletest/.

[Henney02] Kevlin Henney, 'minimalism, the imperial clothing crisis', http://www.two-sdg.demon.co.uk /curbralan/papers/minimalism/TheImperialClothingCrisis.html.

[Henney06] Kevlin Henney, 'Context Encapsulation, Three Stories, a Language, and Some Sequences', January 2006, http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ContextEncapsulation.pdf.

[Henney08] Kevlin Henney, 'The PfA papers: Deglobalisation', *Overload*, February 2008, http://accu.org /index.php/journals/1470.

[Henney09] Kevlin Henney, 'GUT Instinct, Sticky Minds', May 2009, http://www.stickyminds.com /pop_print.asp?ObjectId=14973&ObjectType=ART.

[HR] Hardware register (Wikipedia), http://en.wikipedia.org/wiki/Hardware_register.

[IP] Interface Physics, Universitet Leiden, Netherlands, http://www.physics.LeidenUniv.nl/sections /cm/ip/.

[LPM] Leiden Probe Microscopy, http://www.leidenprobemicroscopy.com/.

[Meszaros07] Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison–Wesley Professional, 2007. See 'Test Spy', http://xunitpatterns.com/Test%20Spy.html.

[Roberts] Tim Roberts, 'If every hardware engineer just understood that...write-only registers make debugging almost impossible', http://www.microsoft.com/whdc/resources /MVP/xtremeMVP_hw.mspx#ETB.

[Rost09] Marcel Rost *et al.*, 'Video-rate Scanning Probe Control Challenges: Setting the Stage for a Microscopy Revolution', *Asian Journal of Control*, March 2009, http://www.physics.leidenuniv.nl/sections /cm/ip/group/PDF/Asian%20J.%20of%20Control/11%282009%29110.pdf.

[SPM] scanning probe microscopy (Wikipedia), http://en.wikipedia.org /wiki/Scanning_probe_microscopy; See also [SPMBBC].

[SPMBBC] scanning probe microscopy (BBC), http://www.bbc.co.uk/dna/h2g2/A717563.

[TC] Tunnel current is the quantum effect that a small current can flow between conductors that have no physical contact if they are a few nm apart, (Wikipedia) http://en.wikipedia.org /wiki/Scanning_tunneling_spectroscopy.

[TDD] Test Driven Development (Wikipedia), http://en.wikipedia.org/wiki/Test-driven_development.

[TeX] inspired on the \relax command of Donald Knuth's TeX typesetting system (Wikipedia), http://en.wikipedia.org/wiki/TeX.

The Model Student: A Game of Six Integers (Part 1)

In how many ways can you combine a set of numbers? Richard Harris gets counting.

Dum de dum de dum, dum de dum de dum, dum de dum de dum, dum de dum de dum, baa daa daa, baa daa daa, boo doo, boo doo, boo doo dee doo, choo.

Ah, Countdown. A nice cup of tea, perhaps a choccy biccy or two, and we're ready to cross mental swords with televisual gladiators in half an hour of orthographical and arithmetical battle.

Er, sorry, it seems I've set the hyperbole switch to turbo.

Hang on a sec.

Righty ho.

Beloved of students, retirees and housewi…, er, home-makers throughout the land, it is one of the longest running game shows in the world and was the very first program broadcast on Channel 4, way back in 1982 [Countdown].

Hosted for much of its astonishingly long run by Carol Vorderman and the splendidly be-blazered and much missed Richard Whiteley, we have entered a new era with the recent departure of the former and the premature demise of the latter.

For the one or two of you who haven't experienced the joy of Countdown, the game consists of two types of rounds; the letters rounds and the numbers rounds.

In the letters rounds, one of the two contestants chooses 9 letters from a random source of consonants and a random source of vowels with which they both seek to construct as long a word as possible.

In the numbers rounds, one of the contestants chooses 6 integers from a concealed set of large integers and a concealed set of small integers with which they both seek an arithmetic formula that yields a value as close as possible to a randomly selected target between 1 and 999.

Guess which one I'm interested in? You got it.

The Countdown numbers game

Before the start of the numbers game the two sets of large and small numbers, printed on cards, are randomly shuffled and placed face down upon a table.

Specifically, the set of large numbers, consisting of one of each of 25, 50, 75 and 100, are placed upon the table above the set of small numbers, consisting of two of each of the integers from 1 to 10, so as to distinguish between them.

One of the contestants then picks 6 of these, typically 1 large and 5 small, before a random target between 100 and 999 is generated.

Using addition, subtraction, multiplication and division they both must then, during a period of just 30 seconds, attempt to discover a formula that yields the target number using each of these numbers no more than once.

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

Furthermore, every intermediate value during the calculation, and hence the result itself, must be a whole number. For example, if the selected numbers were 75, 8, 6, 5, 3, 3 and the target were 234, we would hit it exactly with the formula:

$$75 \times 3 + 6 + 3$$

or, without using the large number, with:

$$5 \times 6 \times 8 - 3 - 3$$

Strictly speaking, a contestant earns points if he or she is the nearest of the pair to the target, provided at least that their result is within a certain maximum difference; let's be honest though, almost is just another word for fail.

Before we can investigate the properties of the numbers game, we shall need some code with which we can enumerate every possible formula that we might construct.

In order to do this, we shall express the formulae in Reverse Polish notation; a spectacularly convenient notation for algorithmically evaluating arithmetic formulae.

Reverse Polish notation

In Reverse Polish notation, or RPN, arithmetic operators are placed immediately to the right of their arguments rather than between them, as in the more familiar infix notation. For example, the formula 3+4 would be expressed as 3 4 + in RPN.

Developed in the late 1950s by Charles Hamblin, it was used to simplify the electronics in early Hewlett-Packard calculators [HP1] and proved so popular that they still support it on some of their models [HP2].

The principal advantages of RPN are that the operators appear in the input sequence precisely when they need to be applied, and that the precedence of the operators is unambiguously determined by their position, eliminating the need for brackets.

We can see this when we express more complex formulae in RPN and to do that we must introduce the RPN stack.

Whenever a number appears in the input sequence of an RPN formula, it is pushed on to the stack. Whenever an operator appears, it pops the arguments it requires off of the stack and pushes the result of the calculation back on to the stack.

An error occurs if there are not enough values on the stack for an operator or if there is more than one value left on the stack at the end of the expression.

For the simple example 3 4 +, we push 3 then 4 on to the stack and the + operator pops them and pushes 7 on to the stack. Since we have reached the end of the expression, this single value on the stack is the result of the calculation.

Using infix notation, we must consider operator precedence to ensure that we arrive at the correct result. For example, the formula

$$2 + 3 \times 4$$

Generating the set of formula templates is a relatively straightforward process

should be interpreted as

$$2 + (3 \times 4)$$

rather than

$$(2 + 3) \times 4$$

since multiplication has higher precedence than addition. If we wish to calculate the latter, we must use brackets to ensure that the operators are applied in the correct order.

In RPN, however, no such complications exist. The first calculation is described by the formula

$$3\ 4 \times 2 +$$

and the second by

$$2\ 3 + 4 \times$$

Running through the steps required to calculate the first of these, we push first 3 then 4 onto the stack. The multiplication operator pops these values off of the stack and pushes their product, 12, back on to it. The value 2 is then pushed onto the stack and the addition operator pops off it and the 12 and pushes back their sum, 14. Since there are no more inputs, this is the result of the calculation.

Figure 1 illustrates graphically the state of the stack after each term in the formula is entered.

Enumerating the RPN formulae

Noting that the four arithmetic operators allowed by the Countdown numbers game, addition, subtraction, multiplication and division, all require precisely two arguments we need only generate RPN formulae consisting of binary operators.

Rather than generate the formulae directly, I propose that we work with a pair of placeholder symbols, *o* and *x*, to represent operators and arguments respectively. We can then subsequently substitute them with every valid permutation of operators and arguments to generate the full set of formulae.

Generating the set of formula templates is a relatively straightforward process. We start with the simplest template, namely a single value *x*. The next step is to replace the single *x* with a binary operation and its arguments, *xxo*, the result of which will also be a single value. Continuing in the same vein, we recursively replace each *x* in turn in the current template with *xxo*, stopping when there are as many *x* symbols as there are available arguments.

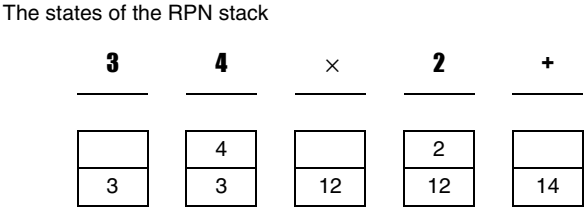

Figure 1

```
std::set<std::string>
all_templates(size_t arguments)
{
  std::set<std::string> result;
  if(arguments!=0)  all_templates("x",
     arguments-1, result);
  return result;
}
```

Listing 1

For example, if we had 3 arguments to work with, we would traverse the following set of formulae.

x
xxo
xxoxo *xxxoo*

In general, some formula templates will be generated more than once during the recursion. For example, if we were working with 4 arguments the template *xxoxxoo* could be generated by replacing with *xxo* both the last *x* in *xxoxo* and the first *x* in *xxxoo* from the 3 argument templates. We will therefore need to keep track of the templates we generate in order not to repeat ourselves.

As it happens, it is rather convenient to use strings of *x* and *o* characters to represent our formula templates. Listing 1 illustrates the `all_templates` function that returns the full set of templates for a given number of available arguments.

This simply forwards the work on to another overload, provided in listing 2.

The key to terminating the recursion when we have exhausted the set of arguments lies in the fact that each time we replace an *x* with an *xxo* we add a single extra argument to the formula template. By passing the remaining number of arguments to the function during the recursion, we can stop when they reach 0.

The main loop iterates over the current template, replacing each *x* with *xxo* in turn and passing each new template recursively to the function for the same treatments.

The complete set of formula templates for up to 5 arguments is illustrated in figure 2, in order of increasing length.

x *xxo* *xxoxo* *xxxoo* *xxoxoxo* *xxoxxoo*
xxxooxo *xxxoxoo* *xxxxooo* *xxoxoxoxo* *xxoxxoxoo* *xxoxxooxo*
xxoxxoxoo *xxoxxxooo* *xxxooxoxo* *xxxooxxoo* *xxxxoooxo* *xxxxoxoxo*
xxxoxxooo *xxxxoooxo* *xxxxooxoo* *xxxxoxooo* *xxxxxoooo*

Figure 2

The formula for the number of equations with up to a given number of arguments is a little more complex

```
void
all_templates(const std::string &current,
              size_t arguments,
              std::set<std::string> &result)
{
  result.insert(current);
  if(arguments!=0)
  {
    std::string::size_type pos = current.find('x');
    while(pos!=std::string::npos)
    {
      std::string next = current;
      next.replace(pos, 1, "xxo");
      if(result.insert(next).second)
      {
        all_templates(next, arguments-1, result);
      }
      pos = current.find('x', pos+1);
    }
  }
}
```

Listing 2

The number of formula templates

One very simple calculation we can perform at this point is to determine the total number of unique formula templates for up to any given number of arguments; we simply call the `size` member function of the set returned by the `all_templates` function.

Note that the difference between the result of this calculation for n arguments and the result for n-1 arguments gives us the number of unique formula templates with exactly n arguments.

Table 1 gives the results of both calculations, which we denote by $T_{1,n}$ and T_n respectively, for 0 to 15 arguments.

n	$T_{1,n}$	Tn
0	0	0
1	1	1
2	2	1
3	4	2
4	9	5
5	23	14
6	65	42
7	197	132

n	$T_{1,n}$	Tn
8	626	429
9	2,056	1,430
10	6,918	4,862
11	23,714	16,796
12	82,500	58,786
13	290,512	208,012
14	1,033,412	742,900
15	3,707,852	2,674,440

Table 1

The number of formulae

From this point it is a relatively simple task to calculate the total number of formulae expressible with a given number of arguments. We shall treat every argument as if it were distinct from all the others since this both dramatically simplifies the calculation and will ultimately yield the correct statistical properties of the Countdown numbers game. Furthermore, we'll restrict ourselves to the 4 binary operators allowed in the numbers game.

Note that we consider formulae distinct even if they could be rearranged to be identical.

To recover the number of formulae with a specific number of arguments, which we shall denote by F_n, we must multiply the number of templates by the number of ways we can replace the o symbols and by the number of ways we can replace the x symbols.

Noting that the templates always contain one less operator than the number of arguments, the first multiplier is given by 4^{n-1}.

The second multiplier is simply the number of orderings of the n arguments, n factorial or $n!$, calculated by multiplying together every number from 1 up to and including n. This hold true since when ordering the n arguments we first pick 1 of the n, then 1 of the remaining n-1, then one of the remaining n-2 and so on.

Hence the number of formulae is given by

$$F_n = T_n \times 4^{n-1} \times n!$$

The formula for the number of equations with up to a given number of arguments is a little more complex, since in this case we must sum the number of ways we can construct formulae with subsets of the arguments.

The number of ways we can select r from n items when the order of selection is important is known as a permutation and is denoted by ${}^nP_r$.

$${}^nP_r = \frac{n!}{(n-r)!}$$

This follows in a similar way to the derivation of the number of orderings of n arguments being equal to $n!$. We first select 1 from the n arguments, then 1 from the remaining n-1 and so on, but this time we stop after we have chosen r of them.

Note that 0! equals 1 and so when r equals n, ${}^nP_r$ is simply $n!$.

The number of formulae with up to n arguments, which we shall denote by $F_{1,n}$ is therefore given by

$$F_{1,n} = \sum_{i=1}^{n} T_i \times 4^{i-1} \times {}^nP_i$$

Recall that the large capital sigma means the sum of the expression to its right with i taking values from 1 up to and including n.

Table 2 gives the results of these calculations for 0 to 15 arguments.

We can now calculate the total number of formulae that might be expressed during the Countdown numbers game by multiplying $F_{1,6}$ by the number of ways we might select the 6 numbers from the 24 on offer. In this case,

the order in which we pick the locations doesn't matter, just the locations themselves

n	$F_{1,n}$	F_n
0	0	0
1	1	1
2	10	8
3	219	192
4	8,500	7,680
5	470,485	430,080
6	33,665,406	30,965,760
7	2,951,054,575	2,724,986,880
8	3.06090E+11	2.83399E+11
9	3.66592E+13	3.40078E+13
10	4.97823E+15	4.62507E+15
11	7.55804E+17	7.03010E+17
12	1.26855E+20	1.18106E+20
13	2.33230E+22	2.17314E+22
14	4.66154E+24	4.34629E+24
15	1.00633E+27	9.38798E+26

Table 2

the order of the number is unimportant and the number is known as a combination, denoted by ${}^nC_r$.

$$ {}^nC_r = \frac{n!}{r! \times (n-r)!} $$

This result follows from the fact that the number of combinations is equal to the number of permutations divided by the number of orderings of the r arguments we have chosen.

The total number of possible formulae is therefore

$$ {}^{24}C_6 \times F_{1,6} = 134,596 \times 33,665,406 $$
$$ = 4,531,228,985,976 $$

So, rather a lot then.

Is there an explicit formula?

Personally, I'd much rather have an explicit, or closed form, formula for T_n than rely upon calling the, as it happens rather computationally expensive, `all_templates` function over and over again. To this end, I spent some time mucking about with a spreadsheet and came up with, for n greater than or equal to 1

$$ T_n = \frac{{}^{2n-1}C_n}{2n-1} $$

This works for every example we've seen so far and, given that the problem is clearly a combinatorial one of some sort or another and that a formula with n arguments has n-1 operators and hence has precisely $2n$-1 terms, it doesn't seem entirely unreasonable.

But, of course, 15 working examples and suspiciously familiar terms are a far cry from an actual proof. Damn you mathematics, you harsh mistress you! Would that you were more like Physics, or better yet Philosophy, so I could get away with any old tosh.

One thing that stands out is that the numerator of the fraction is equal to the number of ways in which one can construct a sequence of n *x* symbols and n-1 *o* symbols, since it is the number of ways in which we can pick the n locations we wish to place *x* symbols. It is a combination, rather than a permutation, since the order in which we pick the locations doesn't matter, just the locations themselves.

This leads to the interpretation that the probability of such a sequence is a valid formula template is equal to

$$ \frac{1}{2n-1} $$

To demonstrate that this is indeed the case we first note that such a sequence is a valid formula template if and only if there have been more *x* symbols than *o* symbols up to and including each and every term.

If this were not so, we would run out of values on the stack at some point, and hence would not have a valid formula template.

Noting that each operator reduces the number of values on the stack by 1, we must therefore have at the end of the calculation a single value on the stack, representing the result of the formula.

Since it implies that we cannot run out of arguments and that we end the calculation with a single value on the stack, this rule ensures that the formula template is valid.

Thankfully, we don't actually need to prove that this rule is obeyed with the presumed probability, since someone has already done it for us.

The Ballot Theorem

The Ballot Theorem [Feller68] was proven in 1878 by W. A. Whitworth and can be stated as

> Suppose that, in a ballot, candidate *P* scores p votes and candidate *Q* scores q votes, where $p>0$ and $p \geq q$. The probability that throughout the counting there are always more votes for *P* than for *Q* equals $(p-q)/(p+q)$.

Our problem is a special case where P represents the x symbols and receives n votes and Q represents the o symbols and receives n-1 votes. According to the Ballot Theorem, the probability that the number of x symbols exceeds the number of o symbols up to and including each and every term is therefore

$$ \frac{n-(n-1)}{n+(n-1)} = \frac{1}{2n-1} $$

as suspected.

We first consider the boundary conditions of p equal to q and of p greater than q and q equal to 0.

In the first case the number of votes must be equal when we finish counting, so the probability that P always has more votes then Q is 0. The formula correctly predicts this, since

$$\frac{p-q}{p+q} = \frac{p-p}{p+p} = \frac{0}{2p} = 0$$

In the second case, the number of votes for P must always exceed the number of votes for Q, since Q hasn't got any, and hence the probability is 1. Again, the formula is in agreement, yielding

$$\frac{p-q}{p+q} = \frac{p-0}{p+0} = \frac{p}{p} = 1$$

The remaining states that we might observe are those where p is greater than q and q is greater than 0. In these cases, we consider the very last vote cast.

If the last vote is for Q, we can treat the penultimate vote as the end of a ballot in which P receives p votes and Q receives q-1 votes, for which we assume that the proposition is true. If, however, it is for P, we treat the penultimate vote as the end of a ballot in which P receives p-1 votes and Q receives q votes, for which we also assume that the proposition is true.

Now, the probability that the last vote cast is for Q is equal to

$$\frac{q}{p+q}$$

and the probability that it is for P is equal to

$$\frac{p}{p+q}$$

Since these two scenarios are independent of each other, the probability that P stays ahead of Q throughout the vote counting is given by

$$\frac{q}{p+q} \times \frac{p-(q-1)}{p+(q-1)} + \frac{p}{p+q} \times \frac{(p-1)-q}{(p-1)+q}$$

Rearranging this, we have

$$\begin{aligned}\frac{q \times (p-(q-1)) + p \times ((p-1)-q)}{(p+q) \times (p+q-1)} &= \frac{(p+q) \times (p-q) + q - p}{(p+q) \times (p+q-1)} \\ &= \frac{(p+q-1) \times (p-q)}{(p+q) \times (p+q-1)} \\ &= \frac{(p-q)}{(p+q)}\end{aligned}$$

We can treat the shorter ballots in exactly the same fashion and must therefore eventually end up with an expression involving a set of ballots in which either the number of votes for P is equal to the number of votes for Q or the number of votes for Q is equal to 0.

Hence the proposition must be true whenever $p>0$ and $p \geq q$, as claimed.

Derivation 1

There are several ways to prove this theorem [Renault07], but to my mind the most elegant by far uses proof by induction, as shown in derivation 1.

What exactly does this have to do with the Countdown numbers game?

Well, er, not very much to be perfectly honest.

That said, the fact that the total number of formulae that can be expressed with up to n distinct variables and the 4 binary operators of addition, subtraction, multiplication and division is equal to

$$F_{1,n} = \sum_{i=1}^{n} \frac{^{2i-1}C_i}{2i-1} \times 4^{i-1} \times P_i$$

is a pretty damn interesting result and, in my opinion at least, was worth a short detour.

Still, I suppose we should probably crack on.

```
template<class T>
class rpn_operator
{
public:
  typedef std::stack<std::vector<T> >  stack_type;
  virtual ~rpn_operator();
  virtual bool apply(stack_type &stack) const = 0;
};
template<class T>
rpn_operator<T>::~rpn_operator()
{
}
```

Listing 3

Implementing RPN operators

The first tool we shall need to assist us in exploring the statistical properties of the Countdown numbers game is a way to transform formula templates into formulae.

We shall begin with an abstract base class to represent an arbitrary RPN operator, called **rpn_operator**, as illustrated in listing 3.

Note that we have defined this as a template class. In general, an RPN calculator would operate upon **double**s, but for the Countdown numbers game we shall need to restrict ourselves to integers. We use the standard **stack** class for the RPN stack since it does everything we need of it, albeit favouring the processing efficiency of **std::vector** to the memory efficiency of **std::deque**.

The return value of the **apply** member function serves to indicate whether the operation yields a valid result for the given arguments. For the integer only numbers game, we should expect at least half of all division operations to yield a fraction and hence be invalid, since if one argument wholly divides, but is not equal to, the other then the latter cannot wholly divide the former. This is a little too common an outcome to be considered truly exceptional, and so we shall be well advised to avoid the relatively expensive C++ exception mechanism.

The declarations and definitions for our RPN operators are given in listing 4. Note that whilst the base class can represent any RPN operator, we are restricting ourselves to the four binary operators used in the numbers game; namely addition, subtraction, multiplication and division.

The definitions of the apply member functions are given in listing 5.

```
template<class T>
class rpn_add : public rpn_operator<T>
{
public:
  virtual bool apply(stack_type &stack) const;
};
template<class T>
class rpn_subtract : public rpn_operator<T>
{
public:
  virtual bool apply(stack_type &stack) const;
};
template<class T>
class rpn_multiply : public rpn_operator<T>
{
public:
  virtual bool apply(stack_type &stack) const;
};
template<class T>
class rpn_divide : public rpn_operator<T>
{
public:
  virtual bool apply(stack_type &stack) const;
};
```

Listing 4

```
template<class T>
inline bool
rpn_add<T>::apply(stack_type &stack) const
{
  if(stack.size()<2)
     throw std::invalid_argument("");
  const T x0 = stack.top();  stack.pop();
  const T x1 = stack.top();  stack.pop();
  stack.push(x0+x1);
  return true;
};
template<class T>
inline bool
rpn_subtract<T>::apply(stack_type &stack) const
{
  if(stack.size()<2)
    throw std::invalid_argument("");
  const T x0 = stack.top();  stack.pop();
  const T x1 = stack.top();  stack.pop();
  stack.push(x0-x1);
  return true;
};
template<class T>
inline bool
rpn_multiply<T>::apply(stack_type &stack) const
{
  if(stack.size()<2)
     throw std::invalid_argument("");
  const T x0 = stack.top();  stack.pop();
  const T x1 = stack.top();  stack.pop();
  stack.push(x0*x1);
  return true;
};
template<class T>
inline bool
rpn_divide<T>::apply(stack_type &stack) const
{
  if(stack.size()<2)
     throw std::invalid_argument("");
  const T x0 = stack.top();  stack.pop();
  const T x1 = stack.top();  stack.pop();
  if(!rpn_valid_divide(x0, x1))  return false;
  stack.push(x0/x1);
  return true;
};
```

Listing 5

The definitions are pretty straightforward; each operator first checks that there are enough arguments on the stack, then pops them and pushes the result back on to the stack. The only complexity is the call to the as yet undefined **rpn_valid_divide** function from **rpn_divide::apply**, which serves to check that the division operation has a valid result.

Illustrated in listing 6, the two overloads distinguish between floating point and integer division. Both return **false** upon division by zero, with latter additionally checking that the second argument wholly divides the first, since this would be an invalid step in the integer-only numbers game.

```
bool
rpn_valid_divide(double x0, double x1)
{
  return x1!=0.0;
}
bool
rpn_valid_divide(long x0, long x1)
{
  return x1!=0 && (x0%x1)==0;
}
```

Listing 6

Of course, we could do a more thorough job with some template wizardry, but these two overloads will suffice for now.

To be perfectly honest, we should probably perform similar tests for numerical overflow and the like during the application of the operators, perhaps returning an error code instead of a boolean so that we might be able to identify exactly what caused a calculation to fail. That said, these sorts of errors are artefacts of the mechanics of computer arithmetic rather than fundamental limitations of arithmetic and as such are of lesser interest.

Evaluating formula templates

All that remains to do before we can evaluate our formula templates is to implement a function that does exactly that. This function needs to parse the formula templates, pushing arguments on to the stack or applying operators as required.

The first thing we need is a return type that can indicate both the validity of the result and its value, as given in listing 7.

If we have a value with which to construct the result it is, by definition, valid and if not it is, by definition, invalid.

Listing 8 provides the definition of the **rpn_evaluate** function with which we shall evaluate our formula template. This function takes a formula template, represented by a **string** containing *x* and *o* characters, a **vector** of pointers to **rpn_operator** objects and a **vector** of argument values and returns an **rpn_result**.

This function simply iterates over the formula template, pushing the current argument on to the stack of the current term is an *x*, and applying the current operator if it is an *o*.

Note that we return an invalid **rpn_result** in the event that any of the intermediate values in the calculation of the formula are invalid.

```
template<class T>
struct rpn_result
{
  rpn_result();
  explicit rpn_result(const T &t);
  bool valid;
  T    value;
};
template<class T>
rpn_result<T>::rpn_result() : valid(false)
{
}
template<class T>
rpn_result<T>::rpn_result(const T &t) :
   valid(true), value(t)
{
}
```

Listing 7

If there aren't enough arguments or operators, or if there is not exactly 1 value on the stack at the end of the calculation, the function throws an exception. Recall that the operators themselves throw exceptions if the stack doesn't contain enough arguments when they are applied.

Listing 9 illustrates how we might use **rpn_evaluate** to calculate the formula 4 2 + 5 ×

If we execute this code, we shall see that the output is 30, as it should be.

Because of the separation of the formula template, the operators and the arguments, the **rpn_evaluate** function isn't particularly amenable for use as a general purpose calculator. It is, however, very well suited to iterating through the set of all possible formulae so that we can examine the statistical properties of their results.

Analysing the Countdown numbers game

We are now very nearly ready to investigate the statistical properties of the numbers game. All that is left for us to do is to work out how to iterate through every possible set of operators and arguments that might be substituted into the formula templates generated by our all_templates function.

We have a clue as to how to go about this in the formula we derived for calculating the total number of possible formulae. Firstly, we shall need a mechanism for enumerating the ${}^{24}C_6$ combinations of 6 numbers from the 24 available. For each of these combinations we shall then need to iterate over the formula templates, setting n to the number of arguments that each in turn requires and enumerating for them the 4n-1 sets of operators and the ${}^{6}P_n$ permutations of arguments.

We shall take as our inspiration the standard next_permutation function with which we can iterate through the set of full permutations of the elements in a given iterator range.

We shall, however, have to wait until next time to transform inspiration into computation, since I have regrettably run out of space in this instalment. So until then, dear reader, do take my advice and, if at all possible, try to catch an episode or two of that most regal of the tea-time quiz fraternity; my beloved Countdown. ■

Acknowledgements

With thank to Keith Garbutt for proof reading this article.

References & Further Reading

[Countdown] http://www.channel4.com/programmes/countdown

[Feller68] Feller, W., *An Introduction to Probability Theory and its Applications*, vol. 1, 3rd ed., Wiley, 1968.

[HP1] http://www.calculator.org

[HP2] http://www.hp.com/calculators

[Renault07] Renault, M., 'Four Proofs of the Ballot Theorem', *Mathematics Magazine*, vol. 80, pp. 345-352, 2007.

```
template<class T>
rpn_result<T>
rpn_evaluate(const std::string &formula,
             const std::vector<rpn_operator<T>
                const *> &operators,
             const std::vector<T> &arguments)
{
  typedef rpn_result<T> result_type;
  typedef typename rpn_operator<T>::
     stack_type stack_type;
  typedef std::vector<rpn_operator<T>
     const *> operators_type;
  typedef std::vector<T> arguments_type;
  std::string::const_iterator
     first_term = formula.begin();
  std::string::const_iterator
     last_term  = formula.end();
  operators_type::const_iterator first_op
     = operators.begin();
  operators_type::const_iterator last_op
     = operators.end();
  arguments_type::const_iterator first_arg
     = arguments.begin();
  arguments_type::const_iterator last_arg
     = arguments.end();
  stack_type stack;
  while(first_term!=last_term)
  {
    switch(*first_term++)
    {
    case 'x':
      if(first_arg==last_arg)
         throw std::invalid_argument("");
      stack.push(*first_arg++);
      break;
    case 'o':
      if(first_op==last_op)
         throw std::invalid_argument("");
      if(!(*first_op++)->apply(stack))
         return result_type();
      break;
    default:
      throw std::invalid_argument("");
    }
  }
  if(stack.size()!=1 || first_op!=last_op
     || first_arg!=last_arg)
  {
    throw std::invalid_argument("");
  }
  return result_type(stack.top());
}
```

Listing 8

```
rpn_add<long> add;
rpn_multiply<long> multiply;
std::vector<rpn_operator<long> const *> ops;
ops.push_back(&add);
ops.push_back(&multiply);
std::vector<long> args;
args.push_back(4);
args.push_back(2);
args.push_back(5);
std::cout
   << rpn_evaluate("xxoxo", ops, args).value
   << std::endl;
std::cout << std::endl;
```

Listing 9

Simplifying the C++/Angelscript Binding Process

Many systems provide a scripting language. Stuart Golodetz shows how to make it easier.

Although it may seem like there's more work involved, there are sometimes significant advantages to be gained by writing your program in more than one language. The specific example I want to highlight is in the field of game development, where, if not ubiquitous, it is certainly commonplace for games to have their artificial intelligence code written in a scripting language rather than in a compiled language like C++. Why is this? The two most significant reasons are that (a) artificial intelligence coding in particular involves a lot of tweaking and tuning – tasks for which compiled languages are not ideally suited – and (b) artificial intelligence code ties in to the game design in a particularly fundamental way, so it is often written by the game designers, who may or may not be experienced programmers. If a game designer wants to implement a simple new feature, and that involves asking an already busy programmer to put it in for them, and a two-day turn-around, then the game as a whole will suffer. A further reason for some games is that the developers want their game to be 'modable', i.e. they want to make it easy for people to modify their game once it's released – whilst many mod writers are experienced programmers, it's a lot easier for people if they just have to modify a script rather than trying to build the entire system.

Incorporating a scripting language into a game is relatively straightforward, but the binding process (i.e. the means by which scripts are allowed to call C++ code, and vice-versa) is often a bit fiddly. This article is about a way I came up with to make the process of adding bindings between C++ and the scripting language I chose to use for my game – AngelScript – a bit less painful. I don't plan to talk about the overall use of AngelScript too much, but the library documentation [AngelScript] is pretty good, and there are tutorials available on the web if you're interested.

Before we start, I'd like to add the disclaimer that comparisons in this article between the original way of doing things in AngelScript and the way I'd prefer are not especially intended as a criticism of the former – AngelScript is more sophisticated as an underlying library than the simple wrapper I'm going to build here really allows for, so in some cases the complexity when using it is a necessary evil. The wrapper I'll describe was originally designed for the specific purpose of making the bits of AngelScript which I found most relevant easier to use – a general solution would take far more work.

Registering C++ functions with AngelScript

One of the first things you usually want to do when you're getting a scripting language up and running in your game is to let your scripts call an in-game function. (For instance, your scripts might need to be able to test line-of-sight between two points in the game world.) To register a function such as `int f(int)` in AngelScript, you have a pointer to a script engine (of type `asIScriptEngine *`), and make a call which looks like:

```
engine->RegisterGlobalFunction("int f(int)",
    asFUNCTION(f), asCALL_CDECL);
```

This seems fairly simple, except for the fact that you have to explicitly specify the AngelScript type of the function as a string, `"int f(int)"`, which is messy (it gets particularly annoying when the function name is longer or the type is more complicated). Ideally, it would be nicer to transform this into something like:

```
myengine->register_global_function(f, "f");
```

The trick to doing this lies in C++'s mechanisms for automatic type deduction. In this instance, it is possible to write, for example, the code in Listing 1:

```
template <typename F>
void ASXEngine::register_global_function(F f,
   const std::string& name)
{
  register_global_function<F>(f, name,
     ASXTypeString<F>(name)());
}
template <typename F>
void ASXEngine::register_global_function(F f,
   const std::string& name,
   const std::string& decl)
{
  int result = m_engine->RegisterGlobalFunction(
     decl.c_str(), asFUNCTION(f), asCALL_CDECL);
  if(result < 0) throw ASXException(
    "Global function " + name + " could not be
    registered");
}
```

Listing 1

Here we get the compiler to deduce the type of the function we're trying to bind for us, then construct the appropriate AngelScript type string using template specialization. All the real work happens in the `ASXTypeString` template. The base template looks like Listing 2.

This is then specialized for (a) simple built-in types like `bool`, `double`, `int`, etc.; (b) const types; (c) pointer and reference types; (d) function types; and (e) function pointer types (see Listing 3).

Note the need to refer to prefix and suffix from `ASXTypeString<T>` as `this->prefix` and `this->suffix` in the above code. This is because `ASXTypeString<T>` is a dependent base class, and non-dependent names are not looked up in dependent base classes in standard C++ [Vandevoorde]. Using `this->` makes the names dependent and causes their lookup to be delayed until the time the template is actually instantiated.

Until C++0x becomes widely implemented, it will be necessary to write specializations like this out long-hand, i.e. a specialization for functions with no arguments, 1 argument, 2 arguments, etc. This works, but is

Stuart Golodetz has been programming for 13 years and is studying for a computing doctorate at Oxford University. His current work is on the automatic segmentation of abdominal CT scans. He can be contacted at stuart.golodetz@comlab.ox.ac.uk

Incorporating a scripting language into a game is relatively straightforward, but the binding process is often a bit fiddly.

```
struct ASXSimpleTypeString
{
  std::string name, prefix, suffix;
  explicit ASXSimpleTypeString(
     const std::string& name_) : name(name_) {}

  std::string operator()() const
  {
    std::ostringstream os;
    if(prefix != "") os << prefix << ' ';
    os << type();
    if(suffix != "") os << ' ' << suffix;
    if(name != "") os << ' ' << name;
    return os.str();
  }
  virtual std::string type() const = 0;
  ASXSimpleTypeString& as_param()
  {
    return *this;
  }
};

template <typename T> struct ASXTypeString :
ASXSimpleTypeString
{
  explicit ASXTypeString(
     const std::string& name_ = "")
     : ASXSimpleTypeString(name_) {}
  std::string type() const {
     return T::type_string(); }
};
```

Listing 2

```
// (a)
// e.g. int
template <>
struct ASXTypeString<int> : ASXSimpleTypeString
{
  explicit ASXTypeString(
     const std::string& name_ = "")
  :  ASXSimpleTypeString(name_)
  {}
  std::string type() const { return "int"; }
};

// (b)
template <typename T>
struct ASXTypeString<const T> : ASXTypeString<T>
```

Listing 3

```
{
  explicit ASXTypeString(
     const std::string& name_ = "")
  : ASXTypeString<T>(name_)
  {
    this->prefix = "const ";
  }
  ASXTypeString& as_param() { return *this; }
};

// (c)
template <typename T>
struct ASXTypeString<T*> : ASXTypeString<T>
{
  explicit ASXTypeString(
     const std::string& name_ = "")
  :  ASXTypeString<T>(name_)
  {
    this->suffix = "@";
  }
};

template <typename T>
struct ASXTypeString<T&> : ASXTypeString<T>
{
  explicit ASXTypeString(
     const std::string& name_ = "")
  :  ASXTypeString<T>(name_)
  {
    this->suffix = "&";
  }
  ASXTypeString& as_param()
  {
    this->suffix = "& out";
    return *this;
  }
};

template <typename T>
struct ASXTypeString<const T&> : ASXTypeString<T>
{
  explicit ASXTypeString(
     const std::string& name_ = "")
  :  ASXTypeString<T>(name_)
  {
    this->prefix = "const";
    this->suffix = "&";
  }
  ASXTypeString& as_param()
  {
    this->suffix = "& in";
```

Listing 3 (cont'd)

So far, this sort of technique is **mildly interesting at best**. It saves us a bit of typing, but nothing more.

```
    return *this;
  }
};

// (d)
// e.g. 1 argument
template <typename R, typename Arg0>
struct ASXTypeString<R (Arg0)>
{
  std::string name;
  explicit ASXTypeString(
     const std::string& name_)
  :  name(name_)
  {}
  std::string operator()() const
  {
    std::ostringstream os;
    os << ASXTypeString<R>()() << ' ' <<
       name << '(';
    os << ASXTypeString<Arg0>().as_param()();
    os << ')';
    return os.str();
  }
};

// (e)
// e.g. 2 arguments
template <typename R, typename Arg0,
   typename Arg1>
struct ASXTypeString<R (*)(Arg0,Arg1)>
   : ASXTypeString<R (Arg0,Arg1)>
{
  explicit ASXTypeString(
     const std::string& name_)
  :  ASXTypeString<R (Arg0,Arg1)>(name_)
  {}
};
```

Listing 3 (cont'd)

extremely tedious – variadic templates will ultimately make this a lot easier.

The type string for a function like `int f(int)` is built up in pieces. At the top level, the `operator()` for an `ASXTypeString<int int)>` is invoked. This invokes the `operator()` of an `ASXTypeString<int>` to get the string `"int"` for the return type of the function. It also invokes the `operator()` of an `ASXTypeString<int>` to get the type string for the argument in this instance, but calls `as_param()` on it first, because things like references translate into different AngelScript types depending on whether they appear as parameters or return types of functions. For example, a `T` reference appearing as a return type should be translated as `"T&"`, whereas one appearing as a parameter should be translated as `"T& out"`. This comes down to the specifics of AngelScript syntax, which are only mildly interesting for the purposes of this article – the key thing is that it's possible (and in this case necessary) to vary the translation depending on where the type actually appears.

```
int funcID
   = module->GetFunctionIdByDecl("int f(int)");
asIScriptContext *context
   = engine->CreateContext();
context->Prepare(funcID);
int arg = 23;
context->SetArgDWord(0, arg);
context->Execute();
int result = context->GetReturnDWord();
context->Release();
```

Listing 4

Calling script functions from C++

So far, this sort of technique is mildly interesting at best. It saves us a bit of typing, but nothing more. It becomes more interesting at the point where we want to call script functions from C++. The normal AngelScript way of doing this for our function `int f(int)` is something like Listing 4.

This works, but it's a lot of hassle just to call a script function. Ideally we'd prefer something like this, where we don't have to specify the full declaration of the AngelScript function, or manually set arguments and retrieve return values:

```
ASXFunction<int(int)> f
   = mymodule->get_global_function("f", f);
int arg = 23;
int result = f(arg);
```

The `get_global_function()` method used above is fairly easy to write. It is also possible to provide an extended version which allows us to still pass in the full declaration of the function. This is useful because there is actually more than one possible way to translate some C++ function types into AngelScript, and we may sometimes wish to explicitly override the default generated by `ASXTypeString` (see Listing 5).

The trick here is to use a dummy parameter to the function, allowing the variable in which the return value is to be stored to be passed in as an argument and its type to be automatically deduced. We've already seen the definition of the `ASXTypeString` template – the rest of the work is done by the `ASXContext` class and `ASXFunction` template. The former is essentially a simple wrapper around `asIScriptContext` (Listing 6).

The `ASXFunction` template and its specializations are more interesting (Listing 7).

The idea here is to wrap the context preparation, argument setting, context execution and value returning together so that we can call script functions without having to worry about the intricate details each time. This is complicated by the fact that the method of setting an argument/retrieving

the method of setting an argument/retrieving a return value when using AngelScript depends fundamentally on the type of the argument

```
template <typename F>
ASXFunction<F> ASXModule::get_global_function(
   const std::string& name,
   const ASXFunction<F>&) const
{
  std::string decl = ASXTypeString<F>(name)();
  int funcID
     = m_module->GetFunctionIdByDecl(
     decl.c_str());
  if(funcID < 0) throw ASXException(
     "Could not find function with declaration "
     + decl);
  asIScriptContext *context
     = m_module->GetEngine()->CreateContext();
  return ASXFunction<F>(ASXContext(context,
     funcID));
}

template <typename F>
ASXFunction<F> ASXModule::get_global_function_ex(
   const std::string& decl,
   const ASXFunction<F>&) const
{
  int funcID
     = m_module->GetFunctionIdByDecl(
     decl.c_str());
  if(funcID < 0) throw ASXException("Could not
     find function with declaration " + decl);
  asIScriptContext *context
     = m_module->GetEngine()->CreateContext();
  return ASXFunction<F>(ASXContext(context,
     funcID));
}
```

Listing 5

```
struct ASXContextReleaser
{
  void operator()(asIScriptContext *context)
  {
    context->Release();
  }
};

ASXContext::ASXContext(asIScriptContext *context,
   int funcID)
:  m_context(context, ASXContextReleaser()),
   m_funcID(funcID)
{
  m_context->SetExceptionCallback(
     asMETHOD(ASXContext, exception_callback),
     this, asCALL_THISCALL);
}

asIScriptContext *ASXContext::operator->() const
{
  return m_context.get();
}

int ASXContext::execute()
{
  return m_context->Execute();
}

int ASXContext::prepare()
{
  return m_context->Prepare(m_funcID);
}

void ASXContext::exception_callback(
   asIScriptContext *context)
{
  int col;
  int row
     = context->GetExceptionLineNumber(&col);

  std::cout  << "A script exception occurred: "
     << context->GetExceptionString() << " at
     position (" << row << ',' << col << ')'
     << std::endl;
}
```

Listing 6 (cont'd)

Listing 6

a return value when using AngelScript depends fundamentally on the type of the argument. For this reason, both are implemented as templates in the above (argument setting is handled by `ASXSetArgValue`, and return value retrieval by `ASXGetReturnValue`). These templates are implemented by writing specializations for the different types we might want to pass in/return (see Listing 8).

With these templates in place, we can then easily acquire a handle to script functions and call them in the usual C++ fashion.

Conclusion

Investing the time to write wrappers like these up-front makes using a scripting language in your program really easy. Of course, there's plenty more we could do – in particular, AngelScript allows you to register C++ types to be used in scripts, and there's a fair amount of work associated with wrapping those sensibly (anyone who is interested is very welcome to email me for the code). The basic idea is much the same, however (and

```
template <typename F> class ASXFunction;

// e.g. 2 arguments
template <typename R, typename Arg0,
   typename Arg1>
class ASXFunction<R (Arg0,Arg1)>
{
private:
  ASXContext m_context;

public:
  explicit ASXFunction(const ASXContext& context)
  :  m_context(context)
  {}

  R operator()(Arg0 value0, Arg1 value1)
  {
    int err = m_context.prepare();
    if(err < 0) throw ASXException(
       "Error preparing script function
       context");

    ASXSetArgValue<Arg0>()(m_context, 0, value0);
    ASXSetArgValue<Arg1>()(m_context, 1, value1);

    err = m_context.execute();
    if(err < 0) throw ASXException(
       "Error executing script function");

    return ASXGetReturnValue<R>()(m_context);
  }
};
```

Listing 7

```
template <typename T> struct ASXSetArgValue
{
  void operator()(const ASXContext& context,
     int arg, T& value) const
  {
    context->SetArgObject(arg, &value);
  }
};

template <typename T> struct ASXSetArgValue<T*>
{
  void operator()(const ASXContext& context,
     int arg, T *value) const
  {
    context->SetArgObject(arg, value);
  }
};

template <> struct ASXSetArgValue<double>
{
  void operator()(const ASXContext& context,
     int arg, double value) const
  {
    context->SetArgDouble(arg, value);
  }
};
template <> struct ASXSetArgValue<int>
{
  void operator()(const ASXContext& context,
     int arg, int value) const
  {
```

Listing 8

```
    context->SetArgDWord(arg, value);
  }
};

template <typename T> struct ASXGetReturnValue
{
  T operator()(const ASXContext& context) const
  {
    return *static_cast<T*>(
       context->GetReturnObject());
  }
};

template <typename T> struct
   ASXGetReturnValue<T*>
{
  T *operator()(const ASXContext& context) const
  {
    return static_cast<T*>(
       context->GetReturnObject());
  }
};

template <> struct ASXGetReturnValue<double>
{
  double operator()(
     const ASXContext& context) const
  {
    return context->GetReturnDouble();
  }
};

template <> struct ASXGetReturnValue<int>
{
  int operator()(const ASXContext& context) const
  {
    return context->GetReturnDWord();
  }
};

template <> struct ASXGetReturnValue<void>
{
  void operator()(const ASXContext&) const
  {}
};
```

Listing 8 (cont'd)

indeed similar ideas can be applied when you're wrapping other scripting languages).

The take-home lessons from this article as a whole are two-fold: firstly, there can be good reasons for developing your program in more than one language, particularly if you need to make it easy for team members who are potentially less experienced to customise functionality without going through you to do it; secondly, it doesn't have to be a particularly painful process. If you think your current project could benefit from scripting, but you're put off because it seems hard to integrate into your existing code, I'd encourage you to take another look. ■

References

[AngelScript] http://www.angelcode.com/angelscript/sdk/docs/manual/index.html

[Vandevoorde] David Vandevoorde and Nicolai M Josuttis, *C++ Templates: The Complete Guide*, pp.136-9.

Quality Matters: Diagnostic Measures

How do we catch problems early? Matthew Wilson investigates the recls library.

This instalment, like the last, involves getting my hands dirty examining another (open-source) library; this time it's **recls** [RECLS], which provides recursive file-system searching via a (largely) platform-independent API. **recls**, which stands for **rec**ursive **ls**, was my first venture into open-source libraries that involved compilation of source (as opposed to pure header-only libraries), and it still bears the scars of the early mistakes I made, so there're rich pickings to be had. (I should also mention that **recls** was the exemplar project for a series of instalments of my former CUJ column, 'Positive Integration', between 2003 and 2005; all these instalments are available online from *Dr Dobb's Journal*; a list of them all is given in http://synesis.com.au/publications.html#columns. I'll attempt as little duplication with them as possible.)

I'll begin with an introduction to recursive search, illustrating why it is such an onerous task using operating system APIs (OS APIs), and give some examples of how it's made easier with **recls**. This will be followed by an introduction to the **recls** architecture: core API, core implementation, and language mappings. The various design decisions will be covered, to give you an understanding of some of the pros and cons to be discussed later.

Then we'll get all 'Software Quality' on it, examining the API, the implementation and the C++ mapping(s). Each examination will cover the extant version (1.8), the new version (1.9) that should be released by the time you read this, and further improvements required in future versions. Naturally, the discussion will be framed by our aspects of software quality [QM#1]: as well as the usual discussions of *intrinsic characteristics*, the problem area – interaction with the file-system and the complexity of the library – dictates the use of *(removable) diagnostic measures* and *applied assurance measures*. It is in the application of the latter two that the meat of this month's learning resides (for me in particular).

Introduction

recls had a proprietary precursor in the dim and distant past, which I originally wrote to obviate the two main issues with recursive file-system search:

- Handling directories: remembering where you are
- Differences in the way file information is obtained between UNIX and Windows

Let's look at a couple of examples to illustrate. Listings 1 and 2 print all files under a given search directory, in UNIX and Windows respectively. Both examples suffer the first issue, since the search APIs yield only the name of the entry (file/directory) retrieved, requiring you to remember the full directory where you have just searched, in order to append each directory name and recurse again.

The second problem can be seen in the extra processing on UNIX. The UNIX search API – `opendir()`/`readdir()` – provides only the file name. To find out whether the entry you've just retrieved is a file or a directory you must issue another system call, `stat()`; you also have to call this to find out file size, timestamps, and so forth. Conversely, the Windows search API – `FindFirstFile()`/`FindNextFile()` - includes all such information in the `WIN32_FIND_DATA` structure that the search functions fill out each time an entry is found.

As I hope both examples clearly illustrate, with either operating system you've got to put in a lot of work just to do a basic search. The mundane preparation of the search directory (appended with the search-all pattern `*.*` in Windows) and the elision of the dots directories – `.` and `..` – dominate the code. And neither of these are terribly good exemplars: I've assumed everything not a regular file is a directory on UNIX, which does not always hold, and I've horribly overloaded the return value of the worker function `list_all_files_r()` to indicate an error condition. More robust versions would do it better, but would include even more code. The intrinsic software evaluations are not all that impressive:

- **Correctness**: Impossible to establish. As defined in the second instalment [QM#2], correctness cannot be established for any library that provides an abstraction over the file-system on a multitasking operating system, so we won't discuss that characteristic further.
- **Robustness**: The size of the code and the fiddly effort work against it.
- **Efficiency**: A moot point with file-system searching, as the disk latency and seek times far outweigh any but the largest inefficiencies in code; interestingly, programs and languages can still have an effect [DDJ-BLOG-RECLS].
- **Portability**: Obviously they're not portable (outside their operating system families); though you can obtain software that emulates the APIs, such as UNIXem [UNIXem] and WINE [WINE].
- **Expressiveness**: Not by any stretch of the term.
- **Flexibility**: The units of currency are C-style strings, `struct dirent`, and `WIN32_FIND_DATA`: no flexibility.
- **Modularity**: No modularity issues.
- **Discoverability**: Pretty good for C APIs, with only two and one data type(s), and four and three system functions, needed for UNIX and Windows, respectively.
- **Transparency**: The transparency of the client code is pretty ordinary.

So let's look at the alternative. Listings 3 and 4 show the same functionality obtained via **recls**' core API, in a step-wise manner (via `Recls_Search()`) and a callback manner (via `Recls_SearchProcess()`) respectively. Listing 5 shows the same functionality obtained via recls C++ mapping (the new unified form available in version 1.9).

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

recls has some unpleasant characteristics, and they're not all addressed yet, even with the latest release

Clearly, each example has benefited from the use of a dedicated library, compared to the first two. Each is more expressive, for three reasons. First, the abstraction level of recursive file-system search has been raised. Second, the evident increased level of portability: indeed none of the examples exhibit any platform-dependencies. Finally, the flexibility of the **recls**' types: note that we can pass entry instances, or their path fields, directly to **FastFormat** [FF-1, FF-2, FF-3]. These factors also contribute to a likely increase in robustness, most particularly in the removal of the fiddly code for handling search directory, dots directories and file information. I'd also argue strongly that the transparency of the code is improved.

On the negative side, modularity has been reduced, since we now depend on **recls** and (albeit indirectly for Listings 3 and 4) on **STLSoft** [STLSOFT].

So, pretty good so far. However, the picture is not perfect. **recls** has some unpleasant characteristics, and they're not all addressed yet, even with the latest release. The purpose of this instalment is to use the flaws in **recls** to illustrate software quality issues involved in writing non-trivial software libraries with unpredictable operating-system interactions. Let's dig in.

The recls library

The recls architecture is comprised of three major parts:

- The core library API (C)
- The core library implementation (C and C++)
- Various language mappings (including C++/STL, C#, COM D, Java, Python, Ruby)

As I've mentioned numerous times previously [QM#3, !(C ^ C++)], I prefer a C-API wherever possible, because it:

- Avoids C++ ABI issues; see Part 2 of *Imperfect C++* [IC++] for more on this
- Tends to be more discoverable, even though it doesn't, in and of itself, tend to engender expressiveness, flexibility or robustness in client code; that's what C++ wrappers are for!
- Allows for interoperability with a wide range of languages.

In the case of **recls**, the interoperability was the clincher, although I'm starting to withdraw from this position somewhat, as I'll discuss later.

The recls core API

The two main entities in **recls** are the *search* and the *entry*. A search comprises a root directory, a search specification, and a set of flags that moderate the search behaviour and the type of information retrieved. An entry is a file-system entry that is found as a result of executing the search at a given time. It provides read-only access to the full path, the drive (on Windows), the directory, the file (name and/or extension), the size (for files), the file-system-specific attributes, the timestamps, as well as other useful pseudo-properties such as search-relative path.

The "search" type

The search type is not visible to client code, and is manipulated as an opaque handle, `hrecls_t`, via API functions. The search type has a state, which is a non-reversible/non-resettable position referring to an item within the directory tree under the given search directory. (Note that the state reflects a localised snapshot: it remembers which file it's on, but what is the next file can change depending on external operating-system action. On a long enumeration it is possible to omit an item that was removed after it commenced and include an item that was not present at the time of commencement, just as is the case with manual enumeration.)

The API functions of concern include:

- `Recls_Search()` – as used in Listing 3.
- `Recls_SearchFeedback()` – same as `Recls_Search()`, plus callback function to notify each directory searched.
- `Recls_SearchClose()` – as used in Listing 3.
- `Recls_GetNext()` – advances the search position without retrieving the details for the entry at the new position.
- `Recls_GetDetails()` – retrieves the details for the entry at the current search position.
- `Recls_GetNextDetails()` – advances the search position and retrieves the details for the entry at the new position.
- `Recls_SearchFtp()` – like `Recls_Search()` but searches FTP servers; Windows-only.

The "entry" type

In contrast, the entry type is only semi-opaque. The API functions that retrieve the entry details from a search handle are defined in terms of the handle type `recls_entry_t` (aka `recls::entry_t` in C++ compilation units), as in:

```
RECLS_API Recls_GetDetails(
  hrecls_t        hSrch
, recls_entry_t*  phEntry
);
```

In the same vein, the API functions that elicit individual characteristics about an entry do so in terms of the handle type, as in:

```
RECLS_FNDECL(size_t) Recls_GetPathProperty(
  recls_entry_t hEntry
, recls_char_t* buffer
, size_t        cchBuffer
);
```

Thus, it is possible to write application code in an operating system-independent manner. However, because different operating systems provide different file-system entry information, and application

Unfortunately, the cheery picture I've painted thus far starts to peel and crack when we look at the implementation

programmers may want access to that information, the underlying type for `recls_entry_t`, `struct recls_entryinfo_t`, is defined in the API (see Listing 6).

You may have noted, from Listing 3, another reason to use the `recls_entryinfo_t struct`: it leads to more succinct code. That's because string access shims [XSTL, FF-2, IC++] are defined for the `recls_strptrs_t` type, as in:

```
# if defined(RECLS_CHAR_TYPE_IS_WCHAR)
inline wchar_t const* c_str_data_w(
# else /* ? RECLS_CHAR_TYPE_IS_WCHAR */
inline char const* c_str_data_a(
# endif /* RECLS_CHAR_TYPE_IS_WCHAR */
  recls_strptrs_t const& ptrs
)
{
return ptrs.begin;
}
# if defined(RECLS_CHAR_TYPE_IS_WCHAR)
inline size_t c_str_len_w(
# else /* ? RECLS_CHAR_TYPE_IS_WCHAR */
inline size_t c_str_len_a(
# endif /* RECLS_CHAR_TYPE_IS_WCHAR */
  recls_strptrs_t const& ptrs
)
{
return static_cast<size_t>(
   ptrs.end - ptrs.begin);
}
```

So when we write

```
ff::fmtln(std::cout, "    {0}", entry->path);
```

the FastFormat application layer [FF-1, FF-2, FF-3] knows to invoke `stlsoft::c_str_data_a()` and `stlsoft::c_str_len_a()` (or the widestring equivalents, in a widestring build) to elicit the string slice representing the path.

Time and size

You may have looked at Listing 6 and wondered about the definitions of `recls_time_t` and `recls_filesize_t`. Here's where the platform-independence falls down. With 1.8 (and earlier), the time and size types were defined as follows:

```
#if defined(RECLS_PLATFORM_IS_UNIX)
typedef time_t          recls_time_t;
typedef off_t           recls_filesize_t;
#elif defined(RECLS_PLATFORM_IS_WINDOWS)
typedef FILETIME        recls_time_t;
typedef ULARGE_INTEGER recls_filesize_t;
. . .
```

The decision to do this was pretty much a fallback, as I didn't think of better alternatives at the time. (If memory serves, the size type results from a time when I was still interested in maintaining compatibility with C++ compilers that did not have 64-bit integer types.) No-one's actually ever complained about this, so either no-one's using time/size information for multi-platform programming or they've learned to live with it. I've learned to live with the size thing by using *conversion shims* [IC++, XSTL] to abstract away the difference between the UNIX and Windows types, as in:

```
ff::fmtln("size of {0} is {1}", entry->path,
   stlsoft::to_uint64(entry->size));
```

But it's still a pain, and a reduction in the transparency of client code. Time is more of a pain, and is considerably less easy to work around.

Both of these detract significantly from the discoverability of the library, and require change. With 1.9 I've redefined `recls_filesize_t` to be a 64-bit unsigned integer, and invoke the conversion shim internally. Alas, I've run out of time with the time attribute, and the inconsistent, platform-dependent time types abide. This will be addressed with 1.10, hopefully sometime later this year.

Intrinsic quality

Let's do a quick check-list of the intrinsic software quality of the core API, and client code that uses it.

- **Robustness**: Robustness is improved due to increased expressiveness and portability.
- **Portability**: Much improved over the OS APIs; time type is still not portable
- **Expressiveness**: Good.
- **Efficiency**: Moot.
- **Flexibility**: Good: entry type and string types all insertable into **FastFormat** (and similar libraries).
- **Modularity**: Dependency on **recls** headers and binaries; C++ mapping also depends on **STLSoft**.
- **Discoverability**: Pretty simple and straightforward API.
- **Transparency**: The transparency of the client code is much improved.

So, from a purely API perspective, clear wins for using **recls** are expressiveness and portability, with some flexibility thrown in the mix.

The recls core implementation

Unfortunately, the cheery picture I've painted thus far starts to peel and crack when we look at the implementation, which is hideously opaque (!transparent).

Implementation language: C or C++?

The first thing to note is that the implementation language is C++. There are two reasons. First, and most significantly, this was so I could use a large number of components from **STLSoft** to assist in the implementation. The main ones are:

there were good reasons for each of these individual steps, but the end result is a big mess

- **`winstl::basic_findfile_sequence`**: for finding directories to navigate the directory tree; for finding files that match a pattern within a given search directory.
- **`inetstl::basic_findfile_sequence`**: for finding files that match a pattern within a given FTP search directory.
- **`unixstl::readdir_sequence`**: for finding directories to navigate the directory tree.
- **`unixstl::glob_sequence`**: for finding files that match a pattern within a given search directory.
- **`platformstl::filesystem_traits`**: for writing path manipulation code in a platform-independent manner.

The other reason was that there is some runtime polymorphism going on inside, allowing for file search and FTP search (Windows-only) to share much of the same surrounding code. Thus, a search begun with **`Recls_SearchFtp()`** can be manipulated in exactly the same way as one begun with **`Recls_Search()`** by client code (and mapping layers). I've long outgrown the perverse pleasure one gets from writing polymorphic code in C, so it had to be C++.

While the first reason did prove itself, in that I was able to implement a large amount of functionality in a relatively short amount of time, I'm not sure that I would do the same again. Some of the code in there is insanely baroque. For example, the constructor of the internal class **`ReclsFileSearchDirectoryNode`** (Listing 7).

This is really, really horrible. As Aussies like to say, 'How embarrassment?'

The class clearly has a large number of member variables; there are member initialiser-list ordering dependencies; even conditionally-compiled different constructors of the member variables! The constructor body contains static assertions to ensure that the member ordering issues do not bite, but that hardly makes up for all the rest of it. Like many codebases, there were good reasons for each of these individual steps, but the end result is a big mess. I can tell you that adding new features to this codebase is a problem.

There are also some per-class memory allocation routines. In particular, the file entry instance type **`recls_entryinfo_t`** (see Listing 6) is of variable length, so that the path, search directory and (for Windows) the short file strings, along with the array of string slices that constitute the directory parts, are all allocated in a single block. This adds further complexity. Unlike the monstrous constructor shown above, however, I would defend this tactic for the entry info. Because it is immutable, and reference-counted (via a hidden prefixed field), it means that all of the complexity involved in dealing with the instances is encapsulated in one place, after which it can be copied easily (via adding a reference) and eventually released via a single call to **`free()`**. I've used this technique many times in the past, and I think it fine. (I may be deluding myself through habit, of course.)

Intrinsic quality

Let's do a quick check-list of the intrinsic software quality of the core implementation.

- **Robustness**: Robustness is kind of anyone's guess, and for the most part has been ironed out due to defects manifesting much higher up in application code; that's not the way to find it!
- **Portability**: Obviously there are platform-specifics contained within the implementation, but it is nonetheless portable across a wide range of UNIX and Windows platforms, so we'd have to concede that it's portability is good. It is not, however, portable to any other kinds of operating systems, and would require work to achieve that.
- **Efficiency**: Moot. I must admit that if you look through the implementation, you can see instances where I've spent effort to achieve performances in the small which are, in all likelihood, irrelevant compared to those of the system calls. Worse, these have compounded the lack of transparency of the code.
- **Expressiveness**: Despite using some pretty expressive components with which to write this, the overall effect in some cases is still overpoweringly complex.
- **Flexibility**: n/a
- **Modularity**: Dependent on **STLSoft** [STLSOFT] (100% header-only). This shouldn't be a problem to C++ programmers.
- **Discoverability**: n/a
- **Transparency**: Pretty poor. My paying job involves a lot of reviewing of other people's code, so it's fair to say this doesn't even come close to the worst I've seen. On the other hand, it doesn't meet the standards for transparency that I advise my clients to adopt, and I would not accept my writing code like this these days.

For anyone who can be bothered to download 1.8 and 1.9, you'll see a lot more files in the `src/` directory for 1.9, as a consequence of my having started to pare away the components from each other. In 1.8, there were sixteen `.cpp` files, and I think I can say that six were good, eight were moderate, and two were bad. The refactoring has helped a lot, such that out of the 21 `.cpp` files in the source directory, eleven are good, eight are moderate, and only two are bad. The numbers back up what I'm trying to do, which is to separate out all parts that are clear and good, or semi-clear and semi-good, in order to reduce the overall cost if/when a full refactoring happens. Of course, as shown above, the bad is still *really* bad. But now the badness is not impinging on the good.

As well as the refactoring reason – letting me see the wood for the trees – there's another reason for splitting up the files, which we'll get to in a minute or two.

The recls C++ mapping(s)

In versions prior to 1.9 **recls** has shipped with two separate mappings for C++ client code:

I've found myself using the C++ mapping for enumeration in commercial projects precisely zero times

- The "C++" mapping, which provides an Iterator [GOF] pattern enumeration interface.
- The STL mapping, which provides STL collections [XSTL], to be consumed in idiomatic STL manner, as shown in Listing 5.

Enumerating with the original C++ mapping would look something like that shown in Listing 8.

The provision of both reflected **recls**' secondary role as a research and writing vehicle for my CUJ column, and also because, at the time (2003), STL was still somewhat novel and unfamiliar to some C++ programmers. In the 6+ years since, I've found myself using the C++ mapping for enumeration in commercial projects precisely zero times, and I've not had much feedback from users making much use of it, either.

So, given that I was already making significant breaking changes, and (temporarily) dropping other mappings, I decided to take the opportunity and merge the best features from the two mappings. Simplistically, the utility functions come from the former "C++" mapping, and the collections come from the former STL mapping.

Consequently, version 1.9 supports only a single C++ mapping, which is comprised of six types:

- **`recls::directory_parts`** – a collection of strings representing the directory parts of a path, e.g. **`["/", "home/", "matthew/"]`** for the path `"/home/matthew/.bashrc"`
- **`recls::entry`** – a type representing all the information about a file-system entry, including path, drive (Windows), directory, file (name and/or extension), size, timestamps, file attributes, search-relative path, and so on.
- **`recls::ftp_search_sequence`** – equivalent to **`recls::search_sequence`** for searching FTP servers (Windows only).
- **`recls::search_sequence`** – a collection of entries matching a search specification and search flags under a given search root.
- **`recls::root_sequence`** – a collection of all the roots on the file-system: always **`["/"]`** for UNIX; all drives on Windows, e.g. **`["B:\", "C:\", "H:\", "I:\", "J:\", "K:\", "L:\", "M:\", "O:\", "P:\", "S:\", "V:\", "W:\", "Z:\"]`** on my current system.
- **`recls::recls_exception`**

and (a growing list; 1.9 is still being polished as I write this) of utility functions:

- **`recls::calculate_directory_size()`** – calculates the total size of all files in the given directory and all its subdirectories.
- **`recls::create_directory()`** – attempts to create a directory, and reports on the number of path parts existing before and after the operation.
- **`recls::combine_paths()`** – combines two path fragments.
- **`recls::derive_relative_path()`** – derives the relative path between two paths.
- **`recls::is_directory_empty()`** – determines whether a directory and all its subdirectories are empty of files.
- **`recls::remove_directory()`** – attempts to remove a directory, and reports on the number of path parts existing before and after the operation.
- **`recls::squeeze_path()`** – squeezes a path into a fixed width for display.
- **`recls::stat()`** – retrieves the entry matching a given path.
- **`recls::wildcardsAll()`** – retrieves the 'search all' pattern for the current operating environment.

Headers

The other change is that now you just include `<recls/recls.hpp>`, which serves two purposes:

- It includes all the headers from all components
- It introduces all the necessary names from the **`recls::cpp`** namespace into the **`recls`** namespace

The result is just a whole lot less to type, or to think about. More discoverable, if you will.

Properties

One other thing to note. In the last chapter (35) of *Imperfect C++* [IC++], I described a set of (largely) portable techniques I'd devised for defining highly efficient properties (as we know them from C# and Ruby) for C++. So, for all compilers that support them (which is pretty much everything better than VC++ 6, which is pretty much everything of import these days), you have the option to elicit entry information via getters, as in

```
std::string srp = entry.get_search_relative_path();
uint64_t    sz  = entry.get_size();
????        ct  = entry.get_creation_time();
// Still platform-dependent ;-/
bool        ro  = entry.is_readonly();
```

or via properties, as in:

```
std::string srp = entry.SearchPelativePath;
uint64_t    sz  = entry.Size;
????        ct  = entry.CreationTime;
// Still platform-dependent ;-/
bool        ro  = entry.IsReadOnly;
```

if you like that kind of thing. (Which I do.)

Quality?

Let's do a quick check-list of the intrinsic software quality of the new C++ mapping.

when we cannot prove correctness we must rely on gathering evidence for robustness

- **Robustness**: Very high: all resources are managed via RAII. Anything that fails does so according to the *Principle Of Most Surprise* [XSTL], via a thrown exception.
- **Portability**: Apart from platform-dependent time type (to be changed in 1.10), it is otherwise portable.
- **Efficiency**: Moot.
- **Expressiveness**: Good.
- **Flexibility**: Excellent. Anything that has a meaningful string form is interpreted via string access shims [XSTL, FF-2, IC++]
- **Modularity**: Dependent on **STLSoft** [STLSOFT] (100% header-only). This shouldn't be a problem to C++ programmers.
- **Discoverability**: Better than either previous mapping ("C++" or STL). Much better than core API. Much, much better than OS APIs.
- **Transparency**: Actually very good. Assuming you understand the principles of STL extension – you've got *Extended STL* [XSTL], right? – and C++ properties – you've got *Imperfect C++* [IC++], right? – then it's very clear, tight, straightforward (see Listing 9). To be honest, looking over it again as I write this I'm amazed how something so neat (nay, might one even say beautiful) could be layered over such an opaque scary mess. I guess that's the magic of abstraction.

Other mappings

I mentioned earlier that interoperability was a major motivator in choosing to provide a C API. In many cases, that's worked really well. For example, I've been able to rewrite the C++ interface for 1.9 with very little concern for changes in the core API between 1.8 and 1.9. The COM mapping was similarly implemented with very little difficulty against the core API; the fact that, in hindsight, I think the COM mapping implementation stinks is immaterial. I'm also pretty happy with the Python and Ruby mappings, although both will definitely benefit from a brush up when I update them to 1.9.

There have been problems with the model however. First, the rather mundane issue that being all in one distribution, every time I update, say the Ruby mapping, I have to release the entire suite of core library and all mappings. This is just painful, and also muddies the waters for users of a subset of the facilities.

Second, and more significantly, with some languages the advantage of not having to reproduce the non-trivial search logic is outweighed by the hassles attendant in writing and maintaining the mapping code, and in distributing the resulting software. The clearest example of this is the .NET mapping. As well as the tiresome P/Invoke issues, a C# mapping requires an underlying C library to be packaged in a separate DLL. On top of the obvious issues to .NET security, the underlying DLL has to managed manually, and one finds oneself still in 'DLL Hell'. (That's the classical version of DLL Hell, not the newer and often more vexing .NET-specific DLL Hell; but that's another story.) As a consequence of these factors, I spent some time last year rewriting **recls** for .NET from scratch, entirely in C#, in part necessitated by some commercial activities. The result, called **recls 100% .NET** [RECLS-100%-NET] was documented in an article I wrote for *DDJ* late last year [DDJ-RECLS]. I may do other rewrites in the future, depending on how well version 1.9 plays with the other language mappings.

Quality assurance

If you remember back to [QM#2], when we cannot prove correctness we must rely on gathering evidence for robustness. A library like **recls**, with admittedly questionable robustness in the core implementation, positively cries out for us to do so.

To hand, we have *(removable) diagnostic measures* and/or *applied assurance measures* ([QM#1]). To save you scrabbling through back issues, I'll reiterate the lists now. (Removable) diagnostic measures can include:

- **Code coverage constructs**
- **Contract enforcements**
- **Diagnostic logging constructs**
- **Static assertions**

Applied assurance measures can include:

- **Automated functional testing**
- **Performance profiling and testing**
- **User acceptance testing (UAT)**
- **Scratch testing**
- **Smoke testing**
- **Code coverage analysis/testing**
- **Review** (manual and automated)
- **Coding standards**
- **Code metrics** (manual and automated)
- Multi-target compilation

Most/all of these can help us with a library like **recls** to reach a point of confidence at which we can 'adjudge [it] to behave according to the expectations of its stakeholders' [QM#2].

First, I'll discuss the items to which the library has been subjected in the past:

- **Contract enforcements.** Though not yet going beyond debug-build assertions, **recls** has been using contract enforcements since its inception.
- **Diagnostic logging.** Until version 1.9, **recls** has had a debug-build-only tracing, to `syslog()` (UNIX) / `OutputDebugString()` (Windows).
- **Static assertions. recls** has used static assertions [IC++] since inception.

Trying keeping a programmer from debugging is like trying to keep a small child from mining its nose

- **Automated functional testing.** Some parts of the library, such as `recls::combine_paths()`, `recls::derive_relative_path()` and `recls::squeeze_path()`, have behaviour that is (wholly or in part) predictable and independent of the system on which they're executed. In version 1.8 (and 1.9), unit tests are included to exercise them. (Note: in version 1.8, some of the squeeze-path tests fail on some edge cases: I've not fixed them because they're not super relevant, they're fixed in 1.9, and I didn't have time to spare!)
- **Performance profiling.** I have done this from time to time, and still do, and it's rare that the (C++) **recls** library performs with any measurable difference from manually written search functions (such as Listings 1 & 2). Surprisingly, the same can't be said for other languages, but that's another story … [DDJ-RECLS-BLOG]
- **Scratch/smoke testing.** Pretty much all the time. Trying keeping a programmer from debugging is like trying to keep a small child from mining its nose.
- **Review.** In my opinion, there's no better microscope of review than writing articles or books about one's own software, and **recls** has had its fair share of that, which has had good effect on the API and on the (1.9) C++ mapping. Bucking the trend, however, is the core implementation, and I assume that's because it's just such a mess.
- **Coding standards.** I have a rigidly consistent, albeit slowly evolving, coding standard, so I think it's reasonable to claim that **recls** has been subject to this effect as much as any commercial code. As the cult of celebrity proves, however, there're plenty of ways to be ugly that aren't immediately apparent.
- **Code metrics.** Until I started compiling this list, it'd never occurred to me to subject the **recls** codebase to my own code analysis tools. As I'm only hours away from giving the Overload editor another bout of dilatory apoplexy, I guess that'll have to wait for another time. I'll try and incorporate it into a wider study of several libraries in a future instalment.
- **Multi-target compilation.** This one's been ticked off from day one, even if much of my UNIX development is done on Windows, using the UNIXem [UNIXem] UNIX emulation library.

On reflection, this is not a bad list, and I guess it helps to explain why **recls** has become the pretty reliable thing it's been for the last 6+ years. As Steve McConnell says 'Successful quality assurance programs use several different techniques to detect different kinds of errors' [CC].

Nonetheless, the coverage is incomplete, occasional defects still occur, and I remain unsure about the behaviour of significant parts of the software under a range of conditions. More needs to be done.

Several measures either have not been used before, or have been used in a limited fashion. The two I believe are now most important are:

- **Code coverage**
- **Diagnostic logging**

Diagnostic logging

I hope you've noticed that many of my libraries work together without actually being coupled to each other. **b64**, **FastFormat**, **Pantheios** [PAN], **recls**, and others work together without having any knowledge of each other. A major reason for this is that they all represent strings as an abstract concept, namely *string access shims* [XSTL, FF-2, IC++]. But that's only a part of it. I think modularity is a huge part of the negative-decision making process of programmers – coupling brings hassle – so much so that I'll be devoting a whole instalment to the subject later this year.

The problem with working with any orthogonal layer of software service such as diagnostic logging, or indeed with any other software component, is that it is a design-time decision that imposes code time, build time and, in many cases, deployment time consequences. Adding diagnostic logging to **recls** would be extremely easy to do by implementing in terms of **Pantheios**, which is a robust, efficient and flexible logging API library, as in:

```
RECLS_API Recls_Stat(
  recls_char_t const* path
, recls_uint32_t      flags
, recls_entry_t*      phEntry
)
{
  pan::log_DEBUG("Recls_Stat(path=", path
    , ", flags="
    , pan::integer(flag, pan::fmt::fullHex)
    , ", ...)");
```

The costs of converting `flags` to a (hexadecimal) string, combining all the string fragments into a single statement, and emitting to the output stream would be paid only if the DEBUG level is enabled; otherwise there's effectively zero cost, on the order of a handful of processor cycles.

Sounds great. The only problem with that is that building and using **recls** would involve one of two things:

- **Pantheios** is bundled with **recls**, and the **recls** build command builds them both. This would increase the download size of **recls** by a factor of four, and increase the build time by about a factor of ten.
- Users would be obliged to separately download and build **Pantheios**, including configuring the **recls**-expected environment variable, before building **recls**. My experience with such situations with other peoples' open source libraries is not encouraging, and I can't imagine most potential users wanting to take that on.

There's the further issue that users may already have their own logging libraries, and prefer to use them to **Pantheios**. (<vainglory>Ok, I'm playing devil's advocate here, since who could imagine such a situation!</vainglory> But the general point stands.)

I think the answer is rather to allow a user to opt-in to a diagnostic logging library if they chose. In C, the only ways to do this are:

- Compile in a dependency on an declared function that is externally defined. This requires the user to define a function such as `recls_logging_callback()`. While this is a viable technique

I hope you get the clear point that the two techniques – code coverage analysis and automated functional testing – are a great partnership in applied quality assurance

when no others suffice, it does leave as many users as not wondering what they've done wrong when they get linker errors the first time they attempt to use your library.

- Provide an API function with which a user can specify a callback at runtime.

I've opted for the second approach. Version 1.9 introduces the new API function `Recls_SetApiLogFunction()`:

```
typedef void (
   RECLS_CALLCONV_DEFAULT   *recls_log_pfn_t)(
  int                  severity
, recls_char_t const* fmt
, va_list              args
);

struct recls_log_severities_t
{
  /** An array of severities, ranked as follows:
    * - [0] - Fatal condition
    * - [1] - Error condition
    * - [2] - Warning condition
    * - [3] - Informational condition
    * - [4] - Debug0 condition
    * - [5] - Debug1 condition
    * - [6] - Debug2 condition
    * Specifying an element with a value <0
      disables logging for that severity.
    */
  int severities[7];
#ifdef __cplusplus
  . . . // ctors
#endif
};

RECLS_FNDECL(void) Recls_SetApiLogFunction(
  recls_log_pfn_t               pfn
, int                           flags
, recls_log_severities_t const* severities
);
```

With this, the user can specify a log function, and a optional list of severity translations. By default, the severity translations are those compatible with **Pantheios**. And `recls_log_pfn_t` just so happens to have the same signature as `pantheios_logvprintf()`, the **Pantheios** (C) API function . But nothing within **recls** depends on, or knows anything about, **Pantheios**, so there's no coupling. You can just as easily define your own API logging function.

Code coverage

Well, I hope you've made it this far, because this is the meat of this instalment. We're going to see some code coverage in action. I'll be using the **xCover** library [XCOVER], which I discussed in a *CVu* article in March 2009 [XCOVER-CVu]. As CVu online is available only to members, non-ACCU members should seriously think about joining this great organisation.

xCover works, for those compilers that support it (VC++ 7+, GCC 4.3+), by borrowing the non-standard `__COUNTER__` pre-processor symbol in marking execution points, and using it to record the passage of the thread of execution through the different branches of the code. At a given sequence point, usually before program exit, the **xCover** library can be asked to report on which execution points have not been executed. In combination with an automated functional test, this can be used to indicate code which may be unused.

Consider the test program in Listing 10, which exercises the functional aspects of the `Recls_CombinePaths()` API function. It's written in C, but the same principle applies to a C++ test program. (If you're interested, the functional testing is done with the **xTests** library [XTESTS], a simple C/C++ unit/component test library that I bundle with all my other open-source libraries).

`XCOVER_REPORT_GROUP_COVERAGE()` is the salient statement. This requests that **xCover** report on all the uncovered marked execution points pertaining to the group "recls.core.extended.combine_paths". This grouping is applied to those parts of the codebase associated with combining paths by using **xCover** constructs. In this way, you divide your codebase logically, in order to support code coverage testing in association with automated functional testing. (You can also request for an overall coverage report, or reports by source file, from within smoke tests, or your running application, as you see fit. It's just that I prefer to associate it with automated functional testing.)

At the moment – and this is why 1.9 is not yet released – I haven't yet got the implementation file refactoring done in such a fashion that the various functionality is properly separated. So, running the test program from Listing 10 with Visual C++ 9 as I write this, I get output along the lines of Figure 1.

All of these are false positives from other core functions defined in the same implementation file: the `Recls_CombinePaths()` function is fully covered by `test.unit.api.combine_paths.c`.

Obviously I've some work to go, and that'll probably also entail adding further refinements to the **xCover** library, to make this work easier. When it's all nicely boxed off, I'll do a proper tutorial instalment about combining code coverage and automated functional testing. Despite the in-progress nature of the technology, I hope you get the clear point that the two techniques – code coverage analysis and automated functional testing – are a great partnership in applied quality assurance. The functional analysis makes sure that whatever you test behaves correctly, and the code coverage analysis makes sure that everything of relevance is tested.

Such things are, as we all know, trivially simple to achieve in other languages (e.g. C#, Java). But despite being harder in C++, they are possible, and we should work towards using them whenever it's worth the effort, as it (almost always) is with a general-purpose open-source library.

```
        ..\..\bin\recls.1.test.unit.api.combine_paths.vc9.mt.exe --verbosity=2
[Start of group recls.core.extended.combine_paths]:
Uncovered code at index 6 in file ../../src/api.extended.cpp, between lines 88 and 483
Uncovered code at index 7 in file ../../src/api.extended.cpp, between lines 88 and 483
. . .
Uncovered code at index 35 in file ../../src/api.extended.cpp, between lines 88 and 483
Uncovered code at index 38 in file ../../src/api.extended.cpp, between lines 502 and 783
. . .
Uncovered code at index 67 in file ../../src/api.extended.cpp, between lines 502 and 783
[End of group recls.core.extended.combine_paths]:
```

Figure 1

Summary

I've examined a well-established open-source library, **recls**, and criticised it in terms of intrinsic quality characteristics, for the core API, core implementation, and the C++ mapping. Where it has come up short I have made adjustments in the forthcoming version 1.9 release, or have identified improvements to be made in subsequent versions.

I have examined the suite of (removable) diagnostic measures and applied assurance measures and have reported on the ongoing work to refine code coverage analysis, in combination with automated functional testing, in the **recls** library, this work to be revisited at a future time in this forum when it is mature. ■

'Quality Matters' online

As mentioned last time, there's a (glacially) slowly developing website for the column – at http://www.quality-matters-to.us/. It now contains some useful links and several definitions from the first three instalments. By the time you read this I hope to have the blog set up. But that's pretty much my web capacity exhausted, so once again I'd ask for any willing ACCU member to offer what I hope would be a small amount of pertinent skills to tart it up, add a discussion board, and what not. Your rewards will be eternal gratitude, endless plaudits, and free copies of my next book. (Or, if you prefer, a promise not to give you free copies of my next book.)

Acknowledgements

As always, my friend Garth Lancaster, has kindly given of his time to read this at the end of a long working week just before my deadline, without complaint (to my manners) and with salient criticisms (of my writing). He does want me to point out that 'How embarrassment?' is a playful part of the Australian vernacular, originating from a comedy show, and is not representative of an endemic poor standard of grammar.

I must also thank, and apologise to, not only Ric Parkin, as editor, but also all his band of reviewers, as I've really pushed them to the wire with my shocking lateness this time. Perhaps Ric will henceforth borrow some wisdom from my wife, and start artificially bringing due dates and times forward in order to effect a magical eleventh hour delivery with time to spare.

References

[!(C ^ C++)] !(C ^ C++), Matthew Wilson, *CVu*, November 2008

[CC] *Code Complete*, 2nd Edition, Steve McConnell, Microsoft Press, 2004

[DDJ-RECLS-BLOG] http://dobbscodetalk.com/index.php?option=com_myblog&show=Recursive-search-examples-pt2-C.html&Itemid=29

[FF-1] 'An Introduction to FastFormat, part 1: The State of the Art', Matthew Wilson, *Overload* 89, February 2009

[FF-2] 'An Introduction to FastFormat, part 2: Custom Argument and Sink Types', Matthew Wilson, *Overload* 90, April 2009

[FF-3] 'An Introduction to FastFormat, part 3: Solving Real Problems, Quickly', Matthew Wilson, *Overload* 91, June 2009

[IC++] *Imperfect C++: Practical Solutions for Real-Life Programming*, Matthew Wilson, Addison-Wesley, 2004

[PAN] http://pantheios.org/

[QM#1] 'Quality Matters, Part 1: Introductions, and Nomenclature', Matthew Wilson, *Overload* 92, August 2009

[QM#2] 'Quality Matters, Part 2: Correctness, Robustness and Reliability', Matthew Wilson, *Overload* 93, October 2009

[QM#3] 'Quality Matters, Part 3: A Case Study in Quality', Matthew Wilson, *Overload* 94, December 2009

[RECLS] http://recls.org/

[STLSOFT] The STLSoft libraries are a collection of (mostly well written, mostly badly documented) C and C++, 100% header-only, thin façades and STL extensions that are used in much of my commercial and open-source programming; available from http://stlsoft.org/

[UNIXem] A simple UNIX emulation library for Windows; available from http://www.synesis.com.au/software/unixem.html

[WINE] http://www.winehq.org/

[XCOVER] http://xcover.org/

[XCOVER-CVu] 'xCover: Code Coverage for C/C++', Matthew Wilson, *CVu*, March 2009; http://accu.org/index.php/journals/c250/

[XSTL] *Extended STL, volume 1: Collections and Iterators*, Matthew Wilson, Addison-Wesley, 2007

[XTESTS] http://xtests.org/

Listings

All numbered listings are at the end of the article:

1. Recursive file search using UNIX's **`opendir()`**/**`readdir()`** API
2. Recursive file search using Windows' **`FindFirstFile()`**/**`FindNextFile()`** API
3. Stepwise recursive file search using **recls**' core API
4. Callback recursive file search using **recls**' core API
5. Recursive file search using **recls**' C++ mapping
6. Definition of **`recls_entryinfo_t`** and associated types
7. Extract of the implementation of **`ReclsFileSearchDirectoryNode`**
8. Example application code using pre-1.9 "C++" mapping
9. Samples of the implementation of the C++ mapping.
10. Unit-test program using xCover for code coverage analysis

```
unsigned list_all_files_r(char const* path)
{
  STLSOFT_ASSERT(NULL != path);
  STLSOFT_ASSERT('\0' != 0[path]);
  std::string directory(path);
  if(directory[directory.size() - 1u] != '/')
  {
    directory += '/';
  }
  DIR* dir = ::opendir(path);
```

Listing 1

```
  if(NULL == dir)
  {
    ff::fmtln(std::cerr,
       "failed to search '{0}': {1} ({2})", path,
       stlsoft::error_desc(errno), errno);
    return ~0u;
  }
  else
  {
    stlsoft::scoped_handle<DIR*>  scoper(dir,
       ::closedir);
    unsigned n = 0u;
    { for(struct dirent* de; NULL != (
       de = ::readdir(dir));)
    {
      if( de->d_name[0] == '.' &&
          de->d_name[1] == '\0')
      {
        // '.'
      }
      else if(de->d_name[0] == '.' &&
              de->d_name[1] == '.' &&
              de->d_name[2] == '\0')
      {
        // '..'
      }
      else
      {
        std::string entryPath
           = directory + de->d_name;
        struct stat st;
        int r = ::stat(entryPath.c_str(), &st);
        if(0 != r)
        {
          ff::fmtln(std::cerr,
             "failed to stat '{0}': {1} ({2})",
             entryPath,
             stlsoft::error_desc(errno), errno);
        }
        else
        {
          if(st.st_mode & S_IFREG)
          {
            ff::fmtln(std::cout, "    {0}",
               entryPath);
            ++n;
          }
          else
          {
            n += list_all_files_r(
               entryPath.c_str());
          }
        }
      }
    }}
    return n;
  }
}
void list_all_files(char const* path)
{
  ff::fmtln(std::cout, "Searching '{0}'", path);
  unsigned n = list_all_files_r(path);
  if(~0u != n)
  {
    ff::fmtln(std::cout, "  {0} file(s) found",
       n);
  }
}
```

Listing 1 (cont'd)

```
unsigned list_all_files_r(char const* path)
{
  STLSOFT_ASSERT(NULL != path);
  STLSOFT_ASSERT('\0' != 0[path]);
  std::string directory(path);
  if(directory[directory.size() - 1u] != '\\')
  {
    directory += '\\';
  }
  std::string searchSpec = directory + "*.*";
  WIN32_FIND_DATA data;
  HANDLE  h = ::FindFirstFile(searchSpec.c_str(),
     &data);
  if(INVALID_HANDLE_VALUE == h)
  {
    DWORD err = ::GetLastError();
    ff::fmtln(std::cerr, "failed to search '{0}'
       : {1} ({2})", path,
       winstl::error_desc(err), err);
    return ~0u;
  }
  else
  {
    stlsoft::scoped_handle<HANDLE>  scoper(h,
       ::FindClose, INVALID_HANDLE_VALUE);
    unsigned n = 0u;
    do
    {
      if(data.dwFileAttributes
         & FILE_ATTRIBUTE_DIRECTORY)
      {
        if( data.cFileName[0] == '.' &&
            data.cFileName[1] == '\0')
        {
          // '.'
        }
        else if(data.cFileName[0] == '.' &&
                data.cFileName[1] == '.' &&
                data.cFileName[2] == '\0')
        {
          // '..'
        }
        else
        {
          n += list_all_files_r((directory
             + data.cFileName).c_str());
        }
      }
      else
      {
        ff::fmtln(std::cout, "    {0}{1}",
           directory, data.cFileName);
        ++n;
      }
    } while(::FindNextFile(h, &data));
    return n;
  }
}
void list_all_files(char const* path)
{
  ff::fmtln(std::cout, "Searching '{0}'", path);
  unsigned n = list_all_files_r(path);
  if(~0u != n)
  {
    ff::fmtln(std::cout,
       "  {0} file(s) found", n);
  }
}
```

Listing 2

```
// Assumes introduction of recls namespace symbols
void list_all_files(char const* path)
{
  ff::fmtln(std::cout, "Searching '{0}'", path);
  hrecls_t    hSrch;
  recls_rc_t  rc  = Recls_Search(path, NULL,
     recls::FILES | recls::RECURSIVE, &hSrch);
  if(RECLS_FAILED(rc))
  {
    ff::fmtln(std::cerr,
       "failed to search '{0}': {1} ({2})",
       path, rc, int(rc));
  }
  else
  {
    stlsoft::scoped_handle<hrecls_t> scoper(
       hSrch, Recls_SearchClose);
    unsigned  n = 0u;
    entry_t   entry;
    { for(RECLS_GetDetails(hSrch, &entry);
       RECLS_SUCCEEDED(rc);
       rc = Recls_GetNextDetails(hSrch, &entry),
       ++n)
    {
      stlsoft::scoped_handle<entry_t> scoper2(
         entry, Recls_CloseDetails);
      ff::fmtln(std::cout, "    {0}",
         entry->path);
    }}
    ff::fmtln(std::cout,
       "  {0} file(s) found", n);
  }
}
```

Listing 3

```
// Assumes introduction of recls namespace symbols
int RECLS_CALLCONV_DEFAULT onFile(
    recls_entry_t               entry
,   recls_process_fn_param_t    param
)
{
  ff::fmtln(std::cout, "    {0}", entry->path);
  ++*static_cast<unsigned*>(param);
  return +1; // continue
}
void list_all_files(char const* path)
{
  ff::fmtln(std::cout, "Searching '{0}'", path);
  unsigned    n   = 0u;
  recls_rc_t  rc  = Recls_SearchProcess(path,
     NULL, recls::FILES | recls::RECURSIVE,
     onFile, &n);
  if(RECLS_SUCCEEDED(rc))
  {
    ff::fmtln(std::cout,
       "  {0} file(s) found", n);
  }
  else
  {
    ff::fmtln(std::cerr,
       "failed to search '{0}': {1} ({2})", path,
       rc, int(rc));
  }
}
```

Listing 4

```
void list_all_files(char const* path)
{
  ff::fmtln(std::cout, "Searching '{0}'", path);
  try
  {
    recls::search_sequence files(path,
       recls::wildcardsAll(), recls::FILES |
       recls::RECURSIVE);
    unsigned n = 0;
    { for(recls::search_sequence
       ::const_iterator i = files.begin();
       i != files.end(); ++i, ++n)
    {
      ff::fmtln(std::cout, "    {0}", *i);
    }}
    ff::fmtln(std::cout,
       "  {0} file(s) found", n);
  }
  catch(recls::recls_exception& x)
  {
    ff::fmtln(std::cerr,
       "failed to search '{0}': {1} ({2})",
       path, x, int(x.get_rc()));
  }
}
```

Listing 5

```
typedef struct recls_entryinfo_t const*
recls_entry_t;

struct recls_strptrs_t
{
  recls_char_t const* begin;
  recls_char_t const* end;
};
struct recls_strptrsptrs_t
{
  struct recls_strptrs_t const* begin;
  struct recls_strptrs_t const* end;
};

#if !defined(RECLS_PURE_API)
struct recls_entryinfo_t
{
  recls_uint32_t              attributes;
  struct recls_strptrs_t      path;
# if defined(RECLS_PLATFORM_IS_WINDOWS)
  struct recls_strptrs_t      shortFile;
  recls_char_t                drive;
# endif /* RECLS_PLATFORM_IS_WINDOWS */
  struct recls_strptrs_t      directory;
  struct recls_strptrs_t      fileName;
  struct recls_strptrs_t      fileExt;
  struct recls_strptrsptrs_t  directoryParts;
# if defined(RECLS_PLATFORM_IS_WINDOWS)
  recls_time_t                creationTime;
# endif /* RECLS_PLATFORM_IS_WINDOWS */
  recls_time_t                modificationTime;
  recls_time_t                lastAccessTime;
# if defined(RECLS_PLATFORM_IS_UNIX)
  recls_time_t              lastStatusChangeTime;
# endif /* RECLS_PLATFORM_IS_UNIX */
  recls_filesize_t            size;
  struct recls_strptrs_t      searchDirectory;
  struct recls_strptrs_t      searchRelativePath;
```

Listing 6

```
  /* Remaining member are undocumented and subject
     to change */
  recls_uint64_t              checkSum;
  recls_uint32_t              extendedFlags[2];
  recls_byte_t                data[1];
};
#endif /* !RECLS_PURE_API */
```

Listing 6 (cont'd)

```
ReclsFileSearchDirectoryNode::
   ReclsFileSearchDirectoryNode(
  recls_uint32_t            flags
, recls_char_t const*       searchDir
, size_t                    rootDirLen
, recls_char_t const*       pattern
, size_t                    patternLen
, hrecls_progress_fn_t      pfn
, recls_process_fn_param_t  param
)
  : m_current(NULL)
  , m_dnode(NULL)
  , m_flags(flags)
  , m_rootDirLen(rootDirLen)
  , m_searchDir()
  , m_searchDirLen(prepare_searchDir_(
    m_searchDir, searchDir))
  , m_pattern(pattern)
  , m_patternLen(patternLen)
  , m_directories(
      searchDir
#if defined(RECLS_PLATFORM_IS_WINDOWS)
    , types::traits_type::pattern_all()
#endif /* platform */
    , dssFlags_from_reclsFlags_(flags))
  , m_directoriesBegin(
      select_iter_if_(
        flags & RECLS_F_RECURSIVE
      , m_directories.begin()
      , m_directories.end()))
  , m_entries(
      searchDir
    , pattern
#ifdef RECLS_SUPPORTS_MULTIPATTERN_
    , types::traits_type::path_separator()
#endif /* RECLS_SUPPORTS_MULTIPATTERN_ */
    , essFlags_from_reclsFlags_(flags))
  , m_entriesBegin(m_entries.begin())
  , m_pfn(pfn)
  , m_param(param)
{
  . . .
}
```

Listing 7

```
void list_all_files(char const* path)
{
  ff::fmtln(std::cout, "Searching '{0}'", path);

  try
  {
    recls::cpp::FileSearch search(path,
       recls::Recls_GetWildcardsAll(),
       recls::FILES | recls::RECURSIVE);

    unsigned n = 0;
    for(; search.HasMoreElements();
       search.GetNext(), ++n)
    {
      recls::cpp::FileEntry entry
        = search.GetCurrentEntry();
      ff::fmtln(std::cout, "    {0}", entry);
    }
    ff::fmtln(std::cout,
       "  {0} file(s) found", n);
  }

  catch(recls::cpp::ReclsException& x)
  {
    ff::fmtln(std::cerr,
       "failed to search '{0}': {1} ({2})",
       path, x, int(x.rc()));
  }
}
```

Listing 8

```
search_sequence::const_iterator
search_sequence::begin() const
{
  hrecls_t    hSrch;
  recls_rc_t  rc = Recls_Search(m_directory,
     m_pattern, m_flags, &hSrch);

  if( RECLS_FAILED(rc) &&
      RECLS_RC_NO_MORE_DATA != rc)
  {
    throw recls_exception(rc);
  }
  return const_iterator(hSrch);
}

ftp_search_sequence::const_iterator
ftp_search_sequence::begin() const
{
  hrecls_t    hSrch;
  recls_rc_t
     rc = Recls_SearchFtp(m_host.c_str(),
     m_username.c_str(), m_password.c_str(),
     m_directory, m_pattern, m_flags, &hSrch);
  if( RECLS_FAILED(rc) &&
      RECLS_RC_NO_MORE_DATA != rc)
  {
    throw recls_exception(rc);
  }

  return const_iterator(hSrch);
}
```

Listing 9

```
template <typename C, typename T, typename V>
basic_search_sequence_const_iterator<C, T, V>&
basic_search_sequence_const_iterator<C, T,
V>::operator ++()
{
  RECLS_MESSAGE_ASSERT(
     "Attempting to increment invalid iterator",
     NULL != m_handle);
  if(RECLS_FAILED(Recls_GetNext(
     m_handle->hSrch)))
  {
    m_handle->Release();
    m_handle = NULL;
  }
  return *this;
}
class entry
{
  . . .
public: /// Attribute Methods
  char_type const* c_str() const
  {
    STLSOFT_ASSERT(NULL != m_entry);
    return m_entry->path.begin;
  }
  . . .
  recls_time_t get_creation_time() const
  {
    STLSOFT_ASSERT(NULL != m_entry);
    return Recls_GetCreationTime(m_entry);
  }
  . . .
  string_type get_path() const
  {
    STLSOFT_ASSERT(NULL != m_entry);
    return string_type(m_entry->path.begin,
       m_entry->path.end);
  }
  string_type get_drive() const
  {
    STLSOFT_ASSERT(NULL != m_entry);
    return string_type(m_entry->path.begin,
       m_entry->directory.begin);
  }
  string_type get_directory_path() const
  {
    STLSOFT_ASSERT(NULL != m_entry);
    return string_type(m_entry->path.begin,
       m_entry->directory.end);
  }
  string_type get_directory() const
  {
    STLSOFT_ASSERT(NULL != m_entry);
    return string_type(m_entry->directory.begin,
       m_entry->directory.end);
  }
  string_type get_file() const
  {
    STLSOFT_ASSERT(NULL != m_entry);
    return string_type(m_entry->fileName.begin,
       m_entry->fileExt.end);
  }
  string_type get_file_name() const
  {
    STLSOFT_ASSERT(NULL != m_entry);
    return string_type(m_entry->fileName.begin,
       m_entry->fileName.end);
  }
```

Listing 9 (cont'd)

```
  string_type get_file_extension() const
  {
    STLSOFT_ASSERT(NULL != m_entry);
    if(m_entry->fileExt.begin
       == m_entry->fileExt.end)
    {
      return string_type();
    }
    else
    {
      return string_type(
         m_entry->fileExt.begin - 1,
         m_entry->fileExt.end);
    }
  }
  . . .
private: /// Member Variables
  recls_entry_t m_entry;
};
```

Listing 9 (cont'd)

```
/* test.unit.api.combine_paths.c */
static void test_1(void);
static void test_2(void);
static void test_3(void);
static void test_4(void);
int main(int argc, char **argv)
{
  int retCode = EXIT_SUCCESS;
  int verbosity = 2;
  XTESTS_COMMANDLINE_PARSEVERBOSITY(argc,
     argv, &verbosity);
  if(XTESTS_START_RUNNER(
     "test.unit.api.combine_paths", verbosity))
  {
    XTESTS_RUN_CASE(test_1);
    XTESTS_RUN_CASE(test_2);
    XTESTS_RUN_CASE(test_3);
    XTESTS_RUN_CASE(test_4);
#ifdef XCOVER_VER
    XCOVER_REPORT_GROUP_COVERAGE(
       "recls.core.extended.combine_paths", NULL);
#endif /* XCOVER_VER */
    XTESTS_PRINT_RESULTS();
    XTESTS_END_RUNNER_UPDATE_EXITCODE(
       &retCode);
  }
  return retCode;
}
. . .
static void test_4()
{
  char    result[101];
  size_t  cch = Recls_CombinePaths("abc", "def",
     &result[0], STLSOFT_NUM_ELEMENTS(result));
  result[cch] = '\0';
  XTESTS_TEST_INTEGER_EQUAL(7u, cch);
#if defined(RECLS_PLATFORM_IS_UNIX)
  XTESTS_TEST_MULTIBYTE_STRING_EQUAL("abc/def",
     result);
#elif defined(RECLS_PLATFORM_IS_WINDOWS)
  XTESTS_TEST_MULTIBYTE_STRING_EQUAL("abc\\def",
     result);
#endif
}
```

Listing 10