overload 120
APRIL 2014 £3

The Rule Of Zero
Updating the C++ Rule Of Three

Teenage Hex
An in-depth look at the attempt to get teenagers interested in programming

Size Matters
How a 64-bit program may (or may not) be better than the same 32-bit program

Quality Matters
How to sort out a codebase that erroneously quenches exceptions

Search with CppCheck
Using static analysis as a powerful code search mechanism

Static: A Force for Good and Evil
How the static keyword in C# can be used and abused, and how to tell the difference

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 120

April 2014

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 121 should be submitted by 1st May 2014 and those for Overload 122 by 1st July 2014.

The ACCU

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

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

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

Copyrights and Trade Marks

Your Life in Their Hands

We leave an increasingly detailed digital footprint. Ric Parkin worries who can see it.

What have you been doing in the last week? Month? Year? Obviously you'll have some vague idea, but not all that detailed.

Let's try just today. Waking up from the phone alarm, you stream the morning news via iPlayer, before checking your email and any missed calls. A commute to work might involve using your Oyster card, or perhaps a trip on a motorway through roadworks with average speed cameras. Work is pretty hectic but you need to look up some things on Stackoverflow and look at a RSA algorithm on github, and manage to make time at lunch to transfer some money between accounts, book a doctor's appointment via the practice website. At least your amazon delivery arrived today, although you had to sign for it. After work, you text a few friends and meet up in a local pub you found after searching on tripadvisor (after phoning your partner to check it's okay of course!), which you pay for with a credit card. Once you get home, you check Facebook and your twitter account, both of which you've updated during the day, and upload some photos of the sunset you saw this evening, tagging them with the time and location automatically. A quick read of some news and politics websites, and it's time for bed.

This is a pretty normal scenario for our lives, day in day out. And most of it will go via an electronic network at some point.

Now imagine that someone could see every single one of those transactions. Not necessarily the detail, but just the basic fact that something happened between you and the other end. I think you could get a pretty good idea of the shape of someone's life, what they did, where they go, who their friends are, what their opinions were.

This could be seen as a touch paranoid, but I have some good reasons to think this is feasible – one of my early jobs involved working on a system to build and visualise networks of information for law enforcement purposes, and a major tool was to import phone call logs to make connections between people. There was also another part that implemented the legal datatrail to record the justification for asking for such data and get it from the telecoms companies. This was a long time ago though, and since then internet connections, mobile phone data (including mast and wifi connections), unsecured web searches (and even secured ones are often logged by the engine), email data, and webcam traffic can all be looked at. See Figure 1 for examples of various visualisation techniques that are used to make sense of these sorts of networks.

And it has been looked at. Not just with a warrant like in my naive youth, but wholesale collection. As we've found out via the Snowden leaks, mass collection of data is now routine [NSA, GCHQ]. Sure, most of the 'uninteresting' stuff will get thrown away pretty soon as it costs too much to look at, and most people aren't of much interest, but it's still there and is seen as a normal thing to do. Although, who knows what's of interest, and what will that be in the future? Even old fashioned intelligence gathering can be remarkably intrusive [Cambridge].

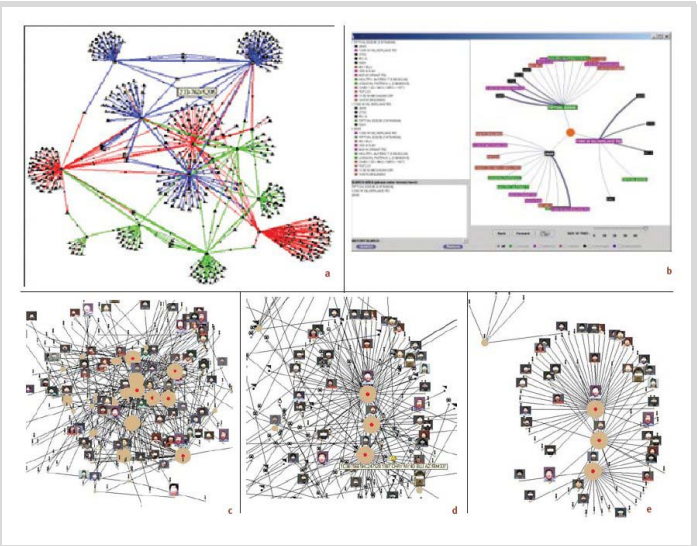


Figure 1

But what of the oft repeated argument that you have nothing to fear if you've nothing to hide? Well, that depends doesn't it? Think of investigative journalists, online legal spats such as Simon Singh and the British Chiropractic Association, climate scientists getting hacked for political purposes, people stalked by new lovers or abused by old, and adopted children tracked down by their unstable relatives, and phone hacking scandals. And what if you knew someone who was interest? Your life has just become of interest itself. And let's not forget when the rules are ignored – there are many known instances of the system being abused by those on the inside.

In the past you could still be tracked if someone really wanted to, but it took time effort people and money. Now it is much, much easier and cheaper, and we now know that the security services have been doing it for quite a while. The lack of oversight, and even complicity, of these activities by the people who are meant to make sure these efforts don't go too far is troubling. Many people on the committees due to their expertise have that because they have close connections to the organisations they are meant to oversee! And often say they cannot tell us why they've agreed to such behaviour because the evidence of why it is useful is itself secret.

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.

But how do we trust them? Some level of access is needed, but we must balance that with the right to privacy and potential for abuse.

What other sorts of data security issues can we think of?

Another potential worry is the NHS.care database. In theory, this is a great idea [Goldacre] – we already get a lot of data from looking at the huge numbers of hospital visits, and good analysis can give us valuable insights into the efficacy of procedures, and identify potentially dangerous hospitals by seeing which operations have a higher than expected mortality or problem rate. By combining this with data from GP visits, we can gain even more insights, especially about how interventions work and detect rare side effects. But the rollout has been badly handled, with poor communication, and worse a poor understanding of people's concerns that even with anonymisation, it can still be easy to discover people and their medical history. Then there were the other purposes to which this data could be put, including potentially being sold to insurance companies – while having better actuarial data could be useful, the worry of being able to tailor insurance to the individual surely defeats the purpose of insurance, which is the pooling of risk. Thankfully this will never happen we are assured, and yet somehow it does [Sold]. This should give us pause for thought [Goldacre2], and has already led to a welcome improvement in transparency [Publish]

At least you're not broadcasting when you're out of your home. Yet. With the upcoming rollout of smart meters [SmartMeters] we are potentially doing just that – while the idea of more accurate metering and billing, not to mention pooling data to enable the energy grid to be better managed, there is a danger that the times you are in or out can be inferred, for example if the meter uploads every half-hour then you can easily work out people's work hours, and even if you choose to only update occasionally, long holidays can be spotted [Frequency]. How can this sort of issue be avoided, while keeping the benefits? One idea would be to have the meters only upload data that can be traced to you only when absolutely required in order to calculate the bill. Other times lots of detailed data could be sent, but only tagged with broad categories and not anything identifiable, although even then you can often work out quite a lot if you can afford the time and effort – this leads to a similar principle to an important rule in cryptography: make the cost of finding out more than it's worth. (Talking of cryptography, it seems that even it may not be enough to be secure, as adding backdoors can weaken it for everybody [Backdoor])

These sorts of situations are becoming more and more frequent, and most people do not have the time, knowledge, or the tools to adequately evaluate and choose what happens to their data, if they even have a choice. Openness and transparency with data collection and storage is important, as are strong checks and balances with an independent watchdog with the ability to investigate abuses and issue strong punishments. But even if we manage to build a good regime in our own country, it is only part of the solution as data is often stored in the cloud, and the datacentres could be in jurisdictions with completely different laws where the hosting company is required to give security agencies access, or can be collected at an intervening point. International rules and co-operation is required with all the problems that are implied by that. But at least some people are thinking about it [Berners-Lee].

References

[Backdoor] http://arstechnica.com/security/2014/01/how-the-nsa-may-have-put-a-backdoor-in-rsas-cryptography-a-technical-primer/

[Berners-Lee] http://www.theguardian.com/technology/2014/mar/12/online-magna-carta-berners-lee-web

[Cambridge] http://www.theguardian.com/uk-news/2013/nov/14/police-cambridge-university-secret-footage

[Goldacre] http://www.theguardian.com/society/2014/feb/21/nhs-plan-share-medical-data-save-lives

[Goldacre2] http://www.theguardian.com/commentisfree/2014/feb/28/care-data-is-in-chaos

[MeterFrequency] http://www.npower.com/idc/groups/wcms_content/@wcms/documents/digitalassets/smart_meter_information_pdf.pdf

[Publish] http://www.theguardian.com/society/2014/mar/07/access-list-national-hospital-records-database-publish

[SmartMeters] https://www.gov.uk/government/policies/helping-households-to-cut-their-energy-bills/supporting-pages/smart-meters

[Sold] http://www.telegraph.co.uk/health/nhs/10659147/Patient-records-should-not-have-been-sold-NHS-admits.html

Size Matters

Should you target 32 or 64 bits? Sergey Ignatchenko and Dmytro Ivanchykhin consider the costs and benefits.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translators and editors. Please also keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, we expressly disclaim all responsibility from any action or inaction resulting from reading this article. All mentions of 'my', 'I', etc. in the text below belong to 'No Bugs' Bunny and to nobody else.

64-bit systems are becoming more and more ubiquitous these days. Not only are servers and PCs 64-bit now, but the most recent Apple A7 CPU (as used in the iPhone 5s) is 64-bit too, with the Qualcomm 6xx to follow suit [Techcrunch14].

On the other hand, all the common 64-bit CPUs and OSs also support running 32-bit applications. This leads us to the question: for 64-bit OS, should I write an application as a native 64-bit one, or is 32-bit good enough? Of course, there is a swarm of developers thinking along the lines of 'bigger is always better' – but in practice it is not always the case. In fact, it has been shown many times that this sort of simplistic approach is often outright misleading – well-known examples include 14" LCDs having a larger viewable area of the screen than 15" CRTs; RAID-2 to RAID-4 being no better compared to RAID-1 (and even RAID-5-vs-RAID-1 is a choice depending on project specifics); and having 41 megapixels in a cellphone camera being quite different from even 10 megapixels in a DSLR despite all the improvements in cellphone cameras [DPReview14]. So, let us see what is the price of going 64-bit (ignoring the migration costs, which can easily be prohibitive, but are outside of scope of this article).

Amount of memory supported (pro 64-bit)

With memory, everything is simple – if your application needs more than roughly 2G–4G RAM, re-compile as 64-bit for a 64-bit OS. However, the number of applications that genuinely needs this amount of RAM is not that high.

Performance – 64-bit arithmetic (pro 64-bit)

The next thing to consider is if your application intensively uses 64-bit (or larger) arithmetic. If it does, it is likely that your application will get a performance boost from being 64-bit at least on x64 architectures (e.g. x86-64 and AMD64). The reason for this is that if you compile an application as 32-bit x86, it gets restricted to the x86 instruction set and this doesn't use operations for 64-bit arithmetic even if the processor you're running on is a 64-bit one.

For example, I've measured (on the same machine and within the same 64-bit OS) the performance of OpenSSL's implementation of RSA, and observed that the 64-bit executable had an advantage of approx. 2x (for RSA-512) to approx. 4x (for RSA-4096) over the 32-bit executable. It is worth noting though that performance here is all about manipulating big numbers, and the advantage of 64-bit arithmetic manifests itself very strongly there, so this should be seen as one extreme example of the advantages of 64-bit executables due to 64-bit arithmetic.

Performance – number of registers (pro 64-bit)

For x64, the number of general purpose registers has been increased compared to x86, from 8 registers to 16. For many computation-intensive applications this may provide a noticeable speed improvement.

For ARM the situation is a bit more complicated. While its 64-bit has almost twice as many general-purpose registers than 32-bit (31 vs 16), there is a somewhat educated guess (based on informal observations of typical code complexity and the number of registers which may be efficiently utilized for such code) that the advantage of doubling 8 registers (as applies to moving from x86 to x64) will be in most cases significantly higher than that gained from doubling 16 registers (moving from 32-bit ARM to 64-bit ARM).

Amount of memory used (pro 32-bit)

With all the benefits of 64-bit platforms outlined above, an obvious question arises: why not simply re-compile everything to 64-bit and forget about 32-bit on 64-bit platforms once and for all? The answer is that with 64-bit, every pointer inevitably takes 8 bytes against 4 bytes with 32-bit, which has its costs. But how much negative effect can it cause in practice?

Impact on performance – worst case is very bad for 64-bit

To investigate, I wrote a simple program that chooses a number *N* and creates a set populated with the numbers 0 to *N*-1, and then benchmarked the following fragment of code:

```
std::set<int> s;
...
int dummyCtr = 0;
int val;
for (j = 0; j < N * 10; ++j)
{
  val = (((rand() << 12) + rand())
        << 12) + rand();
  val %= N;
  dummyCtr += (s.find(val) != s.end());
}
```

When running such a program with gradually increasing *N*, there will be a point when the program will take all available RAM, and will go

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

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He is currently holding the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

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

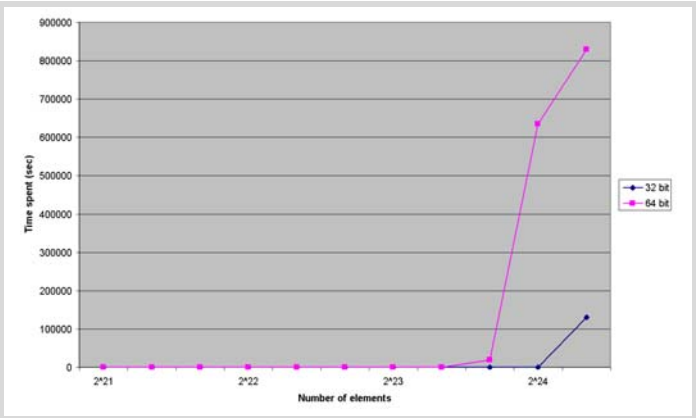


Figure 1

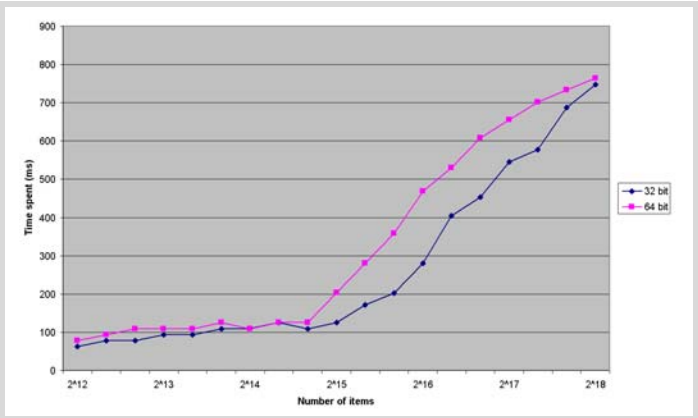


Figure 2

swapping, causing extreme performance degradation (in my case, it was up to 6400x degradation, but your mileage may vary; what is clear, though, is that in any case it is expected to be 2 to 4 orders of magnitude).

I ran the program above (both 32-bit and 64-bit versions) on a 64-bit machine with 1G RAM available, with the results shown on Figure 1.

It is obvious that for N between $2^{24-⅓}$ and $2^{24+⅓}$(that is, roughly, between 13,000,000 and 21,000,000), 32-bit application works about 1000x faster. A pretty bad result for a 64-bit program.

The reason for such behavior is rather obvious: **`set<int>`** is normally implemented as a tree, with each node of the tree[1] containing an **`int`**, two **`bool`**s, and 3 pointers; for 32-bit application it makes each node use 4+4+3*4=20 bytes, and for 64-bit one each node uses 4+4+3*8=32 bytes, or about 1.6 times more. With the amount of physical RAM being the same for 32-bit and 64-bit programs, the number of nodes that can fit in memory for the 64-bit application is expected to be 1.6x smaller than that for the 32-bit application, which roughly corresponds to what we observed on the graph – the ratio of 21,000,000 and 13,000,000 observed in the experiment is indeed very close to the ratio between 32 and 20 bytes.

One may argue that nobody uses a 64-bit OS with a mere 1G RAM; while this is correct, I should mention that [almost] nobody uses a 64-bit OS with one single executable running, and that in fact, if an application uses 1.6x more RAM merely because it was recompiled to 64-bit without giving it a thought, it is a resource hog for no real reason. Another way of seeing it is that if *all* applications exhibit the same behaviour then the total amount of RAM consumed may increase significantly, which will lead to greatly increased amount of swapping and poorer performance for the end-user.

Impact on performance – caches

The effects of increased RAM usage are not limited to extreme cases of swapping. A similar effect (though with a significantly smaller performance hit) can be observed on the boundary of L3 cache. To demonstrate it, I made another experiment. This program:

- chooses a number N
- creates a **`list<int>`** of size of N, with the elements of the list randomized in memory (as it would look after long history of random inserts/erases)
- benchmarks the following piece of code:

```
std::list<int>lst;
...
int dummyCtr = 0;
int stepTotal = 10000000;
int stepCnt = 0;
for (i = 0;; ++i)
{
  std::list <int>::iterator lst_Iter =
    lst.begin();
  for (; lst_Iter != lst.end(); ++lst_Iter)
    dummyCtr += *lst_Iter;
  stepCnt += Total;
  if (stepCnt > stepTotal)
    break;
}
```

With this program, I got the results which are shown in Figure 2 for my test system with 3MB of L3 cache.

As it can be seen, the effect is similar to that with swapping, but is significantly less prominent (the greatest difference in the 'window' from N=2^{15} to N=2^{18} is mere 1.77x with the average of 1.4).

Impact on performance – memory accesses in general

One more thing which can be observed from the graph in Figure 2 is that performance of the 64-bit memory-intensive application in my experiments tends to be worse than that of the 32-bit one (by approx. 10-20%), even if both applications do fit within the cache (or if neither fit). At this point, I tend to attribute this effect to the more intensive usage by 64-bit application of lower-level caches (L1/L2, and other stuff like instruction caches and/or TLB may also be involved), though I admit this is more of a guess now.

Conclusion

As it should be fairly obvious from the above, I suggest to avoid 'automatically' recompiling to 64-bit without significant reasons to do it. So, if you need more than 2–4G RAM, or if you have lots of computational stuff, or if you have benchmarked your application and found that it performs better with 64-bit – by all means, recompile to 64 bits and forget about 32 bits. However, there are cases (especially with memory-intensive apps with complicated data structures and lots of indirections), where move to 64 bits can make your application slower (in extreme cases, orders of magnitude slower). ■

References

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

[Techcrunch14] John Biggs, Qualcomm Announces 64-Bit Snapdragon Chips With Integrated LTE, http://techcrunch.com/2014/02/24/qualcomm-announces-64-bit-snapdragon-chips-with-integrated-lte/

[DPReview14] Dean Holland. Smartphones versus DSLRs versus film: A look at how far we've come. http://connect.dpreview.com/post/5533410947/smartphones-versus-dslr-versus-film?page=4

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

1. for an **`std`** implementation which was used for benchmark testing in this article; differences with other implementations may differ, but not by much

Enforcing the Rule of Zero

We've had years to get used to the old rules for making classes copyable. Juan Alday sees how the new standards change them.

The Rule of Three, considered good practice for many years, became the Rule of Five under C++11. A proper application of the Rule of Five and resource management makes users transition to the Rule of Zero, where the preferable option is to write classes that declare/define neither a destructor nor a copy/move constructor or copy/move assignment operator.

Introduction

The rule of three [Koenig/Moo1] is a rule of thumb coined by Marshall Cline, dating back to 1991. It states that if a class defines a destructor it should almost always define a copy constructor and an assignment operator.

In reality it is two rules:

1. If you define a destructor, you probably need to define a copy constructor and an assignment operator
2. If you defined a copy constructor or assignment operator, you probably will need both, as well as the destructor

Although considered good practice, the compiler can't enforce it. The C++ standard [N1316] mandates that implicit versions will be created if a user doesn't declare them explicitly:

> § 12.4 / 3 If a class has no user-declared destructor, a destructor is declared implicitly
>
> § 12.8 / 4 If the class definition does not explicitly declare a copy constructor, one is declared implicitly.
>
> § 12.8 / 10 If the class definition does not explicitly declare a copy assignment operator, one is declared implicitly

This can lead to code that is fundamentally broken, yet syntactically valid:

```
struct A
{
  A(const char* str) : myStr(strdup(str)) {}
  ~A() {free(myStr);}
private:
  char* myStr;
};
int main()
{
  A foo("whatever");
  A myCopy(foo);
}
```

(Note: Most static analysis tools will detect these errors, but that's beyond the scope of this article)

C++11 [N3242] added wording to the Standard, deprecating the previous behavior.

> D.3 Implicit declaration of copy functions [depr.impldec]
>
> The implicit definition of a copy constructor as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. The implicit definition of a copy assignment operator as defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor.
>
> In a future revision of this International Standard, these implicit definitions could become deleted

This means that compilers keep generating a defaulted copy constructor, assignment operator and destructor if no user-defined declaration is found, but at least now they might issue a warning.

Before and after the adoption of C++11 there were ongoing discussions on the need to ban, rather than deprecate this behavior. C++14 [N3797] kept the same compromise but that does not mean that C++17 will not switch to a full ban [N3839] if adequate wording can be drafted, thus enforcing the need to properly enforce the Rule of Three as a standard of best practices.

C++11 introduced move operations, transforming the Rule of Three into the Rule of Five, implicitly generating move operations under specific circumstances.

There was a lot of controversy regarding automatic generation of an implicit move constructor and assignment operator [Abrahams1] [Abrahams2] [N3153] [N3174] [N3201] [N3203], and the wording was adjusted to reach a compromise and tighten the rules under which they would get defaulted.

C++11 [N3242] says that move operations are ONLY implicitly declared if the following set of conditions occur:

> § 12.8 / 9
>
> If the definition of a class X does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if
>
> - X does not have a user-declared copy constructor,
> - X does not have a user-declared copy assignment operator,
> - X does not have a user-declared move assignment operator,
> - X does not have a user-declared destructor, and
> - The move constructor would not be implicitly defined as deleted.
>
> § 12.8 / 20
>
> If the definition of a class X does not explicitly declare a move assignment operator, one will be implicitly declared as defaulted if and only if
>
> - X does not have a user-declared copy constructor,
> - X does not have a user-declared move constructor,
> - X does not have a user-declared copy assignment operator,
> - X does not have a user-declared destructor, and
> - The move assignment operator would not be implicitly defined as deleted.

Unlike the Rule of three, the Rule of Five is partially enforced by the compiler. You get a default move constructor and move assignment operator if and only if none of the other four are defined/defaulted by the user.

Juan Alday has been working in Wall Street for almost 20 years developing trading systems, mostly in C++. He is a member of accu, acm and pl22.16. He can be reached at juan@greenwiresoft.com

```
struct A
{
  A() : myPtr(API::InitializeStaticData()) {}
  ~A() {API::ReleaseStaticData(myPtr); }
private:
  A(const A&);
  A& operator=(const A&);
  API::Resource* myPtr;
};
```

Listing 1

C++14 expanded the wording, and now an explicit declaration of a move constructor or move assignment operator marks the defaulted copy constructor and assignment operator as deleted. This means that explicit move operations make your objects non-copyable/assignable by default. This is as close as you get to a real enforcement of the rule of five by a compiler.

> § 12.8 / 7
>
> If the class definition does not explicitly declare a copy constructor, one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (8.4). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor.
>
> § 12.8 / 18
>
> If the class definition does not explicitly declare a copy assignment operator, one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy assignment operator is defined as deleted; otherwise, it is defined as defaulted (8.4). The latter case is deprecated if the class has a user-declared copy constructor or a user-declared destructor.

At this point, the Rule of Five transitions in fact to the Rule of Zero, a term coined by Peter Sommerlad [Sommerlad1]:

> Write your classes in a way that you do not need to declare/define neither a destructor, nor a copy/move constructor or copy/move assignment operator
>
> Use smart pointers & standard library classes for managing resources

There are two cases where users generally bypass the compiler and write their own declarations:

1. Managed resources
2. polymorphic deletion and/or virtual functions

Managed resources

> When possible, use a combination of standard class templates like **`std::unique_ptr`** and **`std::shared_ptr`** with custom deleters to avoid managing resources [Sommerlad1]

In C++98/03, a class managing resources would look something like Listing 1. In C++11/14, with move operations, it could be implemented as in Listing 2.

Applying the Rule of Zero, the code would be more expressive (Listing 3).

```
struct A
{
  A() : myPtr(API::InitializeData(),
    &API::ReleaseData) {}
private:
  std::unique_ptr<API::Resource,
    decltype(&API::ReleaseData)> myPtr;
};
```

Listing 3

```
struct A
{
  A() : myPtr(API::InitializeStaticData()) {}
  ~A() {API::ReleaseStaticData(myPtr);}
  A(const A&) =delete; // no need in C++14
  A& operator=(const A&)=delete;
     // no need in C++14
  A(A&& rhs) : myPtr(rhs.myPtr) {
    rhs.myPtr = nullptr;}
  A& operator=(A&& rhs)
  {
    A tmp {std::move(rhs)};
    std::swap(myPtr, tmp.myPtr);
    return *this;
  }

  private:
  API::Resource* myPtr;
};
```

Listing 2

By using a **`unique_ptr`** we make our class non-copyable/assignable and identical in behavior to the previous examples.

`std::unique_ptr` and **`std::shared_ptr`** help us manage pointer types. For non-pointers, Sommerlad and Sandoval [N3949] have proposed two additional RAII wrappers: **`scope_guard`** and **`unique_resource`**, to tie zero or one resource to a cleanup function that gets selectively triggered on scope exit. If it gets accepted, users will have a standard way of managing almost any type of resource automatically.

Users should try to follow this pattern as much as possible and only customize their code when there is no clear alternative. For those cases where we are forced to manage resources (vendor APIs, etc), Martinho Fernandes [Fernandes1] further extends the definition, tying it to the Single Responsibility Principle:

> Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership

Polymorphic deletion / virtual functions

One question on the rule of zero is what to do when we want to support polymorphic deletion, or when our classes have virtual functions:

For years it has been taught that classes supporting inheritance and/or with virtual functions should usually have a virtual destructor. [Stroustrup1] [Koenig/Moo2]

Note: In reality, not all base classes with virtual functions need virtual destructors. Herb Sutter's advice [Sutter1]:

> If A is intended to be used as a base class, and if callers should be able to destroy polymorphically, then make A::~A public and virtual. Otherwise make it protected (and not-virtual)

So a base class with a virtual function like

```
struct A
{
  virtual void foo();
};
```

should be written, following standard practices, as:

```
struct A
{
  virtual ~A() {}
  virtual void foo();
};
```

One side effect is that now both classes are different. After declaring the destructor, **`A`** doesn't support move operations. You still get copy and assignment, but that's as far as you can go.

In C++11 the correct way to define it, in order to get move semantics is:

```
struct A
{
  virtual ~A() =default;
  A(A&&)=default;
  A& operator=(A&&)=default;

  virtual void foo();
};
```

And in C++14 we need to define all five, since otherwise we disable copy/assignment:

```
struct A
{
  virtual ~A() = default;
  A(const A&)=default;
  A& operator=(const A&)=default;
  A(A&&)=default;
  A& operator=(A&&)=default;

  virtual void foo();
};
```

(Note: Depending on your class needs you might also want to add a defaulted constructor, as the implicitly generated default constructor would be marked as deleted since we have specified a copy/move constructor.)

In this case we have applied a consistent rule of five, defaulting all five functions due to the virtual destructor. The question is: Do we really need to do that? Why can't we apply the Rule of Zero?

The second part of the Rule of Zero ("Use smart pointers & standard library classes for managing resources") [Sommerlad1] gives us the answer:

Under current practice, the reason for the virtual destructor is to free resources via a pointer to base. Under the Rule of Zero we shouldn't really be managing our own resources, including instances of our classes (see Listing 4).

We have removed all the default declarations from our base class, and `shared_ptr<A>` will properly invoke `B`'s destructor on scope exit, even though the destructor of `A` is not virtual.

Conclusion

The Rule of Zero lets users do more by writing less. Use it as a guideline when you can and apply the Rule of Five only when you have to.

```
struct A
{
  virtual void foo() = 0;
};
struct B : A
{
  void foo() {}
};

int main()
{
  std::shared_ptr<A> myPtr =
     std::make_shared<B>();
}
```

Listing 4

There is almost no need to manage your own resources so resist the temptation to implement your own copy/assign/move construct/move assign/destructor functions.

Managed resources can be resources inside your class definition or instances of your classes themselves. Refactoring the code around standard containers and class templates like `unique_ptr` or `shared_ptr` will make your code more readable and maintainable.

Help is on its way [N3949] in the form of `scope_guard` and `unique_resource`, to extend the way you can enforce the rule. ■

Acknowledgements

Thanks to Peter Sommerlad, Jonathan Wakely and Ric Parkin for their very helpful comments on drafts of this material.

References

[Abrahams1] Dave Abrahams, Implicit Move Must Go. http://cpp-next.com/archive/2010/10/implicit-move-must-go

[Abrahams2] Dave Abrahams. w00t w00t nix nix http://cpp-next.com/archive/2011/02/w00t-w00t-nix-nix

[Fernandes1] Martinho Fernandes. Rule of Zero. http://flamingdangerzone.com/cxx11/2012/08/15/rule-of-zero.html

[Koenig/Moo1] Andrew Koenig/Barbara Moo. C++ Made Easier: The Rule of Three: http://www.drdobbs.com/c-made-easier-the-rule-of-three/184401400

[Koenig/Moo2] Andrew Koenig/Barbara Moo. *Ruminations in C++. Destructors are special*. ISBN-10 0-201-42339-1

[N1316] Draft for C++03. *Standard for Programming Language C++* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2001/n1316/

[N3153] Dave Abrahams. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3153.htm

[N3174] Bjarne Stroustrup. To move or not to move. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3174.pdf

[N3201] Bjarne Stroustrup. Moving right along. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3201.pdf

[N3203] Jens Maurer. Tightening the conditions for generating implicit moves http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3203.htm

[N3242] Draft for C++11. *Standard for Programming Language C++* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

[N3797] Draft for C++14. *Standard for Programming Language C++* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf

[N3839] Walter Brown. Proposing the Rule of Five, v2. http://open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3839.pdf

[N3949] Peter Sommerlad/A L Sandoval. Scoped Resource – Generic RAII wrapper for the Standard Library http://isocpp.org/files/papers/N3949.pdf

[Sommerlad1] Peter Sommerlad, Simpler C++ with C++11/14, http://wiki.hsr.ch/PeterSommerlad/files/MeetingCPP2013_SimpleC++.pdf

[Stroustrup1] Bjarne Stroustrup. *The C++ Programming Language, Fourth Edition*, section 17.2.5. ISBN-13 978-0-321-56384-2

[Sutter1] Herb Sutter/Andrei Alexandrescu. *C++ Coding Standards*. Ch 50. ISBN-13 978-0-321-11358-0

Quality Matters #8: Exceptions for Recoverable Conditions

Too many programs deal with exceptions incorrectly. Matthew Wilson suggests practical steps to improve your code.

In this instalment I turn to the use of exceptions with (potentially) *recoverable* conditions, and examine what characteristics of exceptions are necessary in order to support recovery. I also consider what may be done when the information to support recovery decisions is not provided, and introduce a new open-source library, **Quench.NET**, that assists in dealing with such situations. (To start with, though, I provide a demonstration of an earlier assertion: exceptions are essentially nothing more than a flow-control mechanism and we should be careful not to forget it.)

Introduction

This is the eighth instalment of Quality Matters, the fourth on exceptions, and the first since I averred in the April 2013 edition of *Overload* that I would be both regular and frequent from here on in. The reason I'd been so sketchy 2010–2013 was that I'd been on a very 'heavy' engagement as an expert witness on an all-consuming intellectual copyright case. Since then, I've been doing double duty as a development consultant/manager/software architect for a large Australian online retailer and working on a start-up, the latter of which is now occupying me full time (which means 70+hr weeks, as is the way of these things…). Both of these later activities have informed significantly on the issues of software quality in general and exception-handling in particular, and so I come back full of ideas (if not *completely* fresh and full of energy) and I intend to get back to more writing and even meet the odd publishing deadline.

In this instalment, I consider a little more about what exceptions are, how they're *used* in *recoverable situations*, and what characteristics are required specifically to facilitate the discrimination between *recoverable* and *practically-unrecoverable* conditions. This is a much bigger sub-subject than I'd ever anticipated when I started on the topic those several years ago, and I now expect to be covering recoverability alone for several instalments. I won't make any further predictions beyond that. I examine problems both with the design and use of exceptions (and exception hierarchies) in reporting failure, and the misapprehension regarding and misuse of thrown exceptions by programmers. I finish by introducing a new open-source library, **Quench.NET**, that is designed to assist with the hairy task of retrofitting fixes to the former, and tolerating the latter.

Prelude of the pedantic contrarian

As I expressed at the end of the first instalment on exceptions, exceptions are not an error-reporting (or, more properly, a failure-reporting) mechanism per se; *all* they are intrinsically is a flow-control mechanism. It just so happens that they can be (and, indeed, were conceived to be and should be) used as a failure-reporting mechanism. But just because that's how they're often used does not mean that that's what they *are*. I've been surprised lately just how hard a concept this is to convey to programmers, so I'm going to hammer it home with a simple sample (see Listing 1).

Now, there are a whole lot of simplifications and compromises in this code for pedagogical purposes (such as use of `private-set` properties rather than immutable fields – *yuck!*) since I'm trying to make my point in the smallest space. (The fact that I *cannot* force myself to skip the failure-handling code adds greatly to the evident size of this simple code, but I hope you understand why I cannot do so, gentle reader.)

I believe it's self-evident that the `FindCompletionException` type is used exclusively within the *normative* behaviour of the program. It is not used to indicate failure; rather it is used specifically to indicate that the program has achieved its aim, which is to find and report the location, size and modification time of the first file matching the given name. Should I choose to do so, I *could* employ this tactic in real production code, and the program would not be in any sense ill-formed for the baroque nature of its implementation. (I confess that I have used exceptions as a short-cut normative escape from deep recursion, but it was a very, very long time

Exception-class Split Personality Syndrome

`System.ArgumentException` serves double duty, both as a specific exception class that is used to report specific conditions, and as a base class of a group of (argument-related) exception classes. This overloading of purpose raises two problems.

First, it means that one must be careful to remember to include the required derived types in a 'multiple catch handler block' in order to catch the specific types, and to order them correctly (most specific classes first).

Second, and more importantly in my opinion, it makes the intent (of the type) ambiguous:

- Am I catching `ArgumentException` as `ArgumentException`, or am I catching all 'argument exceptions'?; and
- Why does condition A have its own specific exception type (e.g. `System.ArgumentOutOfRangeException`) and condition B does not and instead uses the general one `ArgumentException`)?

For example, why can we not have an `ArgumentInvalidValueException`, maybe even an `ArgumentEmptyException`, and so forth?

The same problem occurs with `System.IO.IOException` and its parsimonious stable of derived classes – you get a `FileNotFoundException` and a `DirectoryNotFoundException` ok, but if some failures that occur when trying to access a network path yield a plan-old `IOException`! Unlike argument exceptions, I/O exceptions are likely to be involved in making decisions about recoverability, so this design flaw has much more serious consequences.

Matthew Wilson is a Yorkshire-Australian who cannot help but apply his innate contrarianism, stubbornness, tenacity, and other-bewilderment to his life's passions as author, cyclist, drummer, manager, programmer, trainer, ... with mixed results. Matt's recent career change sees him as a founding partner and Director of Software Architecture for Hamlet Research, a company aiming to change the way we use the world's resources. He can be contacted at matthew.wilson@hamletresearch.net.

what characteristics are required specifically to facilitate the discrimination between recoverable and practically-unrecoverable conditions

```
namespace ExceptionFlowControl
{
  using System;
  using System.IO;

  class FindCompletionException
    : Exception
  {
    public string FileDirectoryPath
      { get; private set; }
    public DateTime FileDate { get; private set; }
    public long FileSize { get; private set; }

    public FindCompletionException(
      string   directoryPath
    , DateTime dateTime
    , long     size
    )
    {
      FileDirectoryPath = directoryPath;
      FileDate = dateTime;
      FileSize = size;
    }
  }

  static class Program
  {
    private static void FindFile
       (string directory, string fileFullName)
    {
      System.IO.DirectoryInfo di =
         new DirectoryInfo(directory);
      foreach(FileInfo fi in
         di.EnumerateFiles(fileFullName))
      {
        if(fi.Name == fileFullName)
        {
          throw new FindCompletionException
           (fi.DirectoryName
          , fi.LastWriteTime, fi.Length);
        }
      }
      foreach(DirectoryInfo subdi in
        di.EnumerateDirectories())
      {
        FindFile(subdi.FullName, fileFullName);
      }
    }

    static void Main(string[] args)
    {
      if(2 != args.Length)
      {
        Console.Error.WriteLine("USAGE:
          program <root-directory><file>");
        Environment.Exit(1);
      }

      Environment.ExitCode = 1;
      try
      {
        FindFile(args[0], args[1]);
        Console.Error.WriteLine("could not find
          '{0}' under directory '{1}'",
          args[1], args[0]);
      }

      catch(FindCompletionException x)
      {
        Console.WriteLine("found '{0}' in '{1}':
          it was modified on {1} and is {2}
          byte(s) long", args[1],
          x.FileDirectoryPath, x.FileDate,
          x.FileSize);

        Environment.ExitCode = 0;
      }

      catch(Exception x)
      {
        Console.Error.WriteLine("could not find
          '{0}' under directory '{1}': {2}",
          args[1], args[0], x.Message);
      }
    }
  }
}
```

Listing 1

Listing 1 (cont'd)

ago, when I was only a neophyte C++ programmer, just out of college ... for shame! :$)

'Please note, still-gentle reader, that I'm not saying this is good design, or a good use of exceptions; indeed, I'm saying it's poor design, and a foolish use of exceptions. But it does illustrate what is possible, and in a way that is not a perversion of the language or runtime.

'Since I have become incapable of writing code that is knowingly wrong/inadequate – especially in an article espousing quality – I have perforce included a `catch(Exception)` handler to deal with failing conditions

```
. . .
static void Main(string[] args)
{
  Environment.ExitCode = 1;

  try
  {
. . . // same
  }
  catch(IndexOutOfRangeException)
  {
    Console.Error.WriteLine("USAGE: program
      <root-directory><file>");
  }
  catch(FindCompletionException x)
  {
. . . // same
  }
  catch(Exception x)
  {
. . . // same
  }
}
```

Listing 2

(such as specification of an invalid directory). Consequently, this program not only illustrates my main point that *exceptions* are a *flow-control mechanism*, but also makes a secondary illustration that the use of the exceptions is overloaded – in this case we're doing both short-circuiting normative behaviour *and* failure-handling.

Had I been so foolish I could have gone further in overloading the meaning applied to the use of exceptions by eschewing the **`if`**-statement and instead catching **`IndexOutOfRangeException`** and issuing the USAGE advisory there, as shown in Listing 2. Alas, I've seen exactly this perversion in production code recently.

The problems of such a choice are described in detail in [QM#5]; in summary:

- **`IndexOutOfRangeException`** is taken by many programmers to be an indication of programmer error, not some valid runtime condition; and, given that
- In a more complex program there may be a bona fide coding error that precipitates **`IndexOutOfRangeException`**, which is then (mis)handled to inform the surprised user that (s)he has misused the program.

Thus, we would see a conflation of (and confusion between) three meanings of the use of exceptions:

- for short-circuit processing (*normative*);
- for reporting runtime failure (*recoverable* and, as in this case, *practically-unrecoverable*); and
- for reporting programmer error (*faulted*).

Recoverable

When we looked at the business of handling *practically-unrecoverable* conditions [QM#5], we utilised two aspects of the exceptions that were being used to report those conditions. First, we relied on the type of the exception to indicate generally the broad sweep of the failure: a **`std::bad_alloc`** was interpreted as an out-of-memory condition in the runtime heap; a **`clasp::clasp_exception`** indicated that an invalid command-line argument (combination) had been specified by the user; a **`recls::recls_exception`** indicated that a file-system search operation had failed.

A second-level of interpretation was then done in some of those handlers to inform (the user, via contingent reports, the administrator/developer/power-user via diagnostic log-statements) the nature of the failure by utilising *state* 'attributes' of the exception types: the unrecognised

```
string portName = . . .
int baudRate = . . .
Parity parity = . . .
int dataBits = . . .
StopBits stopBits = . . .
byte[] request = . . .
byte[] response = . . .

try
{
  SerialPort port = new SerialPort(portName,
    baudRate, parity, dataBits, stopBits);
  port.Open();
  port.Write(request, 0, request.Length);
  int numRead = port.Read(response, 0,
    response.Length);
  . . .
}

catch(Exception x)
{
  Console.Error.WriteLine("exception ({0}): {1}",
    x.GetType().FullName, x.Message);
  Environment.Exit(1);
}
```

Listing 3

command-line option/flag for a **`clasp_exception`**; the missing/inaccessible item for a **`recls_exception`**.

Information can be conferred from the site of failure to the site of handling in three ways. Two are obvious: **type** and **state**; the third is **context**, which can be so obvious as to escape observation.

Type, state, and context

In the examples examined in [QM#5], only *type* was used in discriminating between how to handle the *practically-unrecoverable* conditions, and only insofar as it determined what type-specific *state* information could be presented in the diagnostic log statement and contingent report in addition to (or, in some cases, instead of) the **`what()`**-message information provided by every C++ exception class (that derives from **`std::exception`**).

Handling (potentially) *recoverable* conditions requires more than this simple and somewhat vague recipe. This is because a decision has to be made by a chunk of decision-making software coded by we humans, and since (non-faulting) software only does what we tell it to do (however indirectly and/or inchoately), we are going to need information with a whole bunch of exacting characteristics.

Before I define them – and I won't provide a fully-fledged definition until a later instalment – I would like to consider an example drawn from recent practical experience, which illustrates nicely issues of *type*, *state*, and *context*. Consider the C# code in Listing 3, simplified for pedagogical purpose. There are four important normative actions to consider:

1. Construct an instance of the port (**`System.IO.Ports.SerialPort`**);
2. Open the port instance;
3. Write the request to the port;
4. Read the response from the port.
5.

The constructor of **`System.IO.Ports.SerialPort`** has the following signature:

```
SerialPort(
  string    portName
, int       baudRate
, Parity    parity
, int       dataBits
, StopBits  stopBits
);
```

`Open()` has no parameters and a return type of `void`. `Read()` and `Write()` have the following signatures:

```
int
Read(
  byte[]  buffer
, int     offset
, int     count
);

void
Write(
  byte[]  buffer
, int     offset
, int     count
);
```

There are numerous ways in which this set of statements can fail, ranging from programmer error through to hardware failures, and they can be said to fall into one of two failure classes: *unacceptable arguments*, and *runtime failures*.

Unacceptable arguments

This class of failures pertain to 'unacceptable arguments', and comprises predominantly those that arise from conditions that could, in theory, be dealt with by other means (without incurring thrown exceptions); in some cases failure to do so can be deemed reasonably to be programming error; in others a judgement has to be made as to the best manner to deal with the possible condition; in only a minority is an exception the only credible alternative. The findings are summarised in Table 1.

Null port name; null request; null response

If `portName` is null, the constructor throws an instance of `System.ArgumentNullException` (which derives from `System.ArgumentException`) with the `ParamName` property value (`string`) `"PortName"`, and the `Message` property (string) value `"Value cannot be null.\r\nParameter name: PortName"`.

When considering the constructor alone, this is definitive, because no other parameter of the constructor can (conceivably) throw this exception. If we consider all four statements, though, it is clear that an `ArgumentNullException` can also come from `Read()` and `Write()`: if request is null, the `Write()` method throws an instance of `ArgumentNullException`, with `ParamName` value `"buffer"` and the `Message` value `"Buffer cannot be null.\r\nParameter name: buffer"`; if `response` is null, the `Read()` method throws an instance of `ArgumentNullException` with *exactly* the same properties.

Were we to need to identify programmatically the specific source of the problem given the `try-catch` block structure of Listing 3, we would not be able to discriminate between null argument failure to `Write()` and `Read()` at all, and to do so between either of these methods and the constructor would require reliance on the value of the `ParamName` property. Thankfully, we do not have to worry about this, since it is hard to conceive of any scenario where we might want to wrest recovery from such a circumstance. In my opinion, passing a null value for `portName`, request, or response is a programming error, plain and simple, and all thoughts of recovery should be forgotten.

Empty port name

If `portName` is the empty string, `""`, the constructor throws an instance of `ArgumentException`, with the `ParamName` value `"PortName"`, and the `Message` value `"The Portname cannot be empty.\r\nParameter name: PortName"`.

Invalid port name

If `portName` is `"C:\Windows"`, the constructor completes, but the `Open()` method throws an instance of `ArgumentException`, with the `ParamName` value `"portName"`, and the `Message` value `"The given port name does not start with COM/com or does not resolve to a valid serial port.\r\nParameter name: portName"`.

Zero baud rate

If baudRate is 0, the constructor throws an instance of `System.ArgumentOutOfRangeException`, with the `ParamName` value `"BaudRate"`, and the `Message` value `"Positive number required.\r\nParameter name: BaudRate"`.

Negative baud rate

If `baudRate` is -1, the behaviour is exactly the same as with a zero baud rate.

Zero data bits

If `dataBits` is 0, the constructor throws an instance of `ArgumentOutOfRangeException`, with the `ParamName` value `"DataBits"` and the `Message` value `"Argument must be between 5 and 8.\r\nParameter name: DataBits"`.

Negative data bits; out-of-range data bits

If `dataBits` is -1, or any number outside the (inclusive) range 5–8, the behaviour is exactly the same as with a zero data bits.

Condition	Method	Exception	ParamName
Null Port Name	`ctor`	`System.ArgumentNullException`	`PortName`
Null Request	`Write()`	`System.ArgumentNullException`	`buffer`
Null Response	`Read()`	`System.ArgumentException`	`buffer`
Empty Port Name	`ctor`	`System.ArgumentException`	`PortName`
Invalid Port Name	`Open()`	`System.ArgumentException`	`portName`
Zero Baud Rate	`ctor`	`System.ArgumentOutOfRangeException`	`BaudRate`
Negative Baud Rate	`ctor`	`System.ArgumentOutOfRangeException`	`BaudRate`
Zero Data Bits	`ctor`	`System.ArgumentOutOfRangeException`	`DataBits`
Negative Data Bits	`ctor`	`System.ArgumentOutOfRangeException`	`DataBits`
Out-of-range Data Bits	`ctor`	`System.ArgumentOutOfRangeException`	`DataBits`
Unnamed Parity Values	`ctor`	`System.ArgumentOutOfRangeException`	`Parity`
Unnamed StopBits Values	`ctor`	`System.ArgumentOutOfRangeException`	`StopBits`
Invalid Buffer Length	`Read()`, `Write()`	`System.ArgumentException`	(null)

Table 1

Unnamed parity values

If `parity` is `(Parity)(-1)`, the constructor throws an instance of `ArgumentOutOfRangeException`, with the `ParamName` value `"Parity"` and the `Message` value `"Enum value was out of legal range.\r\nParameter name: Parity"`.

It has the same behaviour for other unnamed `Parity` values [ENUMS].

Unnamed StopBits values

The constructor exhibits the same behaviour as for unnamed values of `Parity` (except that the parameter name is `"StopBits"`).

Invalid buffer length

If we specify, say, `response.Length + 10` in our call to `Read()` (and having written enough such that it will attempt to use the non-existent extra 10), then we will be thrown an instance of `ArgumentException`, with `ParamName` value null and the `Message` value `"Offset and length were out of bounds for the array or count is greater than the number of elements from index to the end of the source collection."`.

Analysis

Unlike the case with null port name, it may be argued that empty and invalid port names are legitimate, albeit practically-unrecoverable, runtime conditions: a user may have entered either in a dialog, or they may be obtained via configuration file settings (that can be wrong). Clearly, being able to filter out an empty name at a higher level is both easy and arguably desirable, and it depends on the details of your design as to whether you choose to do so.

It is clear that zero and negative baud rates are incorrect, and specification of either could be stipulated to be programmer error (and therefore could have been prevented outside the purview of this class, i.e. in the client code's filtering layer). There are several widely-recognised baud-rates, but, as far as I know, there is no final definitive list of serial port baud rates. Hence, we have to be able to supply a (positive) integer value, and we need to be able to account both for reading this from somewhere (e.g. config., command-line) at runtime and for the fact that the value presented may be rejected by the device (e.g. as being outside its minimum/maximum range).

In most respects, data-bits may be considered in the same way as baud rate, just that the possible range is (by convention) small and finite, i.e. between five and eight.

Where things get more interesting are in the two enumeration type parameters, `parity` and `stopBits`. By providing an enumeration, the API implies that the set of options is small, known exhaustively, and fixed, and that there should be no way to specify programmatically an invalid value. Furthermore, since .NET has reasonably good (not great!) facilities for helping interconvert between enumeration values and strings, we should be able to rely (at the level of abstraction of `SerialPort`) on receiving only valid values.

There are three important specific points to make:

1. The port name is not validated (completely) until the port is opened. I suggest that this is reasonable, albeit not necessarily desirable;
2. The value of the `ParamName` property, `"portName"`, differs from that in the first two cases, where it was `"PortName"`. Clearly there's an inconsistency in the implementation, and I suggest that this means we cannot rely on the values contained in `ParamName` as being definitive (which I doubt anyone would be contemplating anyway). I further suggest that we must distrust any framework exception string properties as having accurate, precise, and reliable values;
3. While it's easy to imagine how it may have come to be implemented this way, it is nonetheless inconceivable to me that a parameter-less method – `Open()` in this case – can throw an instance of `ArgumentException` (or any of its derived types)! Perhaps I'm being precious, but I find this tremendously confidence-sapping when faced with using this component.

Runtime failures

This class of failures comprises those that arise entirely through circumstances in the runtime environment, and could be experienced by even the 'best designed program' (whatever that means). The findings are summarised in Table 2.

Unknown (but valid) port name

If `portName` is `"COM9"`, which does not exist on my system, the constructor completes but the `Open()` method throws an instance of `System.IO.IOException`, and the `Message` (string) property has the value `"The port 'COM9' does not exist."`; the `InnerException` property is `null`.

The HRESULT value associated with the exception is 0x80131920, which is COR_E_IO, as documented in MSDN for `System.IO.IOException`. Note that this is not available via any public field/property/method, and requires reflection to elicit (when possible).

Device disconnected before write

In the case where `portName` is of valid form and corresponds to an existing and available port on the current system, the call to `Open()` may return (indicating success). If the port device becomes subsequently unavailable – e.g. by depowering it or unplugging the device from the computer – a call to `Write()` results in the throwing of an instance of `System.IO.IOException`, with the `Message` value `"The device does not recognize the command."`; the `InnerException` property is `null`.

The HRESULT value associated with the exception is 0x80070016, which is no well-known constant (of which I'm aware) in its own right. However, since it uses the well-known `FACILITY_WIN32` (7), the lower 16-bits should correspond to a Windows 'error' code (defined in `WinError.h`). The Windows constant `ERROR_BAD_COMMAND` (22L == 0x16) is associated with the message `"The device does not recognize the command."`, so that seems like our culprit. Clearly, some .NET exceptions carry Windows failure codes (wrapped up in HRESULT values).

Device disconnected before read

In the case where the `Write()` succeeds but the device then becomes unavailable, the subsequent call to `Read()` results in the throwing of an instance of `System.InvalidOperationException`, with the `Message` value `"The port is closed."`; the `InnerException` property is `null`.

The HRESULT value associated with the exception is 0x80131509, which is COR_E_INVALIDOPERATION, as documented in MSDN for `System.InvalidOperationException`. Note that, just as with `IOException`, this is not available via any public field/property/method, and requires reflection to elicit (when possible).

Condition	Method	Exception	(HResult)	Message
Unknown Port Name	Open()	System.IO.IOException	0x80131920	"The port 'COM9' does not exist."
Device Disconnected Before Write	Write()	System.IO.IOException	0x80070016	"The device does not recognise the command."
Device Disconnected Before Read	Read()	System.InvalidOperationException	0x80131509	"The port is closed."

Table 2

Furthermore, in some circumstances (e.g. when performing reads on another thread via `SerialPort`'s `DataReceived` event, which may have 'first bite' at the device failure, resulting in a change of state, such that) a call to `Write()` may also result in a thrown `InvalidOperationException`, rather than `IOException`.

Analysis

There are clear problems presented by these runtime failures for arbitrating successfully between recoverable and practically-unrecoverable conditions, and for providing useful information in any diagnostic log statements/contingent reports in either case.

Perhaps the least important problem is that two of the three messages border on the useless:

- `"The port is closed"` is not likely to enlighten the normal or power user much beyond the self-evident insight that 'the system is not working for me';
- `"The device does not recognise the command"` sounds like the basis of something useful, but it neglects to specify (or even hint at) which command: does it mean the .NET class method just invoked, or the underlying system device command. Whichever, it seems bizarre since the device must at some other times recognise the 'command', so isn't the real message that '*the device cannot fulfil the command* <COMMAND> *in the current state*', or some such;
- Only the message `"The port 'COM9' does not exist"` is adequate, insofar as it will provide something genuinely meaningful to whomever will read the diagnostic log/contingent report

All the remaining problems are more serious, and reflect what I have experienced to be a very poor standard of design in the .NET exception hierarchy and of application of the exception types in other components, particularly those that interact with the operating system.

I also note here that the `Exception.Message` property documentation is unclear, particularly with respect to localisation, which means we cannot rely on the message contents with any precision such as we would need were we to think of, say, parsing information in exception messages in order to make recoverability decisions. The information contained therein can only be relied upon to be potentially helpful in a diagnostic log statement / contingent report.

First, although I've mentioned somewhat casually the HRESULT values associated with the exceptions, these are not publicly available. They are stored in a private field `_HResult` within the `Exception` type, which is read/write-accessible to derived types via the `HResult` protected property. Some exception classes, such as `IOException`, provide the means to supply an HRESULT value in the constructor that directly pertains to the condition that precipitated the exception; when not provided a stock code is used that represents the subsystem or exception class (e.g. COR_E_IO for `IOException`), as seems always the case for those exception types that do not provide such a constructor (e.g. COR_E_INVALIDOPERATION for `InvalidOperationException`).

The only way to access this value is to use reflection, whether directly (e.g. using `Object.GetType()`, `Type.GetField()`, `FieldInfo.GetValue()`) or via `System.Runtime.InteropServices.Marshal.GetHRForException()`. In either case, this will succeed only in execution contexts in which the calling thread has the requisite rights. Absent that, we cannot assume we'll be able to access the value, which pretty much kills it for general-purpose programming.

And it gets worse. The meaning ascribed to the HRESULT value is not consistent. In the above three cases it is condition-specific only in the **Device Disconnected Before Write** case, which is reported by an instance of `IOException`. In the other two cases it is sub-system/exception-class specific, one reported by `InvalidOperationException` and the other by `IOException`! We cannot expect condition-specific information even within one (`IOException`-rooted) branch of the `Exception` family tree.

Third, given that (i) .NET does not have checked exceptions and documentation must always be taken with a grain of salt, and (ii) I've discovered the above behaviour through testing, which will necessarily be inexhaustive, we must pose the obvious question: *How do we know there aren't others?* Indeed, the documentation states that: `Open()` may also throw `System.UnauthorizedAccessException`; `Read()` may also throw `System.TimeoutException`; `Write()` may also throw `System.ServiceProcess.TimeoutException`. Note the two different `TimeoutException` types. If the documentation is correct, it's bad design, which does not engender confidence. If the documentation is wrong, it's bad documentation, which does not engender confidence.

Imagine now, if you will, how we might actually use the serial port in the wild. In one of the daemons that we're developing we are interfacing to an external hardware device via a serial port. The device runs continually, and the daemon must (attempt to) maintain connectivity to it on an ongoing basis. In such a case it is essential to be able to react differently to a practically-unrecoverable condition, such as the wrong port name being specified in the daemon configuration information (**Unknown Port Name**), and a (potentially) recoverable loss of connectivity to the device (**Device Disconnected ...**). In the former case, we want to react to the reported condition by issuing diagnostic and contingent report information and terminating the process (since there's absolutely no sense in continuing); in the latter we want the daemon to issue diagnostic information (which will raise alarms in the wider system environment) but retry continually (until it reconnects or until a human tells it to stop). In order to write that system to these requirements, we need to be able to distinguish between the failures.

Thus, the final and most compelling, disturbing, and disabling problem is that the .NET `SerialPort` component does not support our eminently sensible failure behaviour requirements, because the **Unknown Port Name** condition and the **Device Disconnected ...** conditions are reported by the same type of exception, `IOException`, whose messages we must not parse (since we do not know if we can trust them), and whose putatively definitive discriminating HRESULT information is assigned inconsistently and may not even be accessible!

There is no way, with the given type and state information provided, to discriminate between these two conditions.

Contrast this with the situation in C, programming with the Windows API functions `CreateFile()`, `WriteFile()`, and `ReadFile()`: if we pass an unknown COM port we get `ERROR_FILE_NOT_FOUND`; if we pass `"C:\Windows"` we get `ERROR_ACCESS_DENIED`; if we have a comms failure we get `ERROR_BAD_COMMAND`. It's certainly arguable that the latter two constants' names do not directly bear on the conditions that precipitated them, but the point is that programming at the C-level allows us to discriminate these conditions by state, relying on accessible and (as far as I can tell) predictable and reliable values; programming at the C#-level (using the .NET standard library) does not.

Resort to context

This only leaves context. Our only recourse is to wrap separately the calls to `Open()` and `Write()` in `try-catch` in order to intercept the useless-in-a-wider-context exceptions and translate them into something definitive, along the lines shown in Listing 4.

Let's be clear about this: what we're trying to do with the serial port is a combination of good practices – simplicity, abstraction, transparency – in so far as we're focusing on the normative code, and relying on the 'sophistication' of exception-handling to allow us to deal with failures elsewhere. Unfortunately, the .NET exception hierarchy is poorly designed and badly applied, so we've been forced to pollute the code with low-level `try-catch` handlers, using *context* to rebuild the missing *type*/*state* information before passing on the reported failure condition; it's arguable that using P/Invoke to get at the Windows API calls and using return codes would have been better, which is quite a reversal!

We've had to violate one of the important purposes/advantages of the exception-paradigm, the separation of normative code from failure-handling code, for the purposes of increased transparency and

```
SerialPort port = new SerialPort(portName,
  baudRate, parity, dataBits, stopBits);

try
{
  port.Open();
}
catch(IOException x)
{
  throw new UnknownPortNameException(portName,
    x);
}

try
{
  port.Write(request, 0, request.Length);
}
catch(IOException x)
{
  throw new PortWriteFailedException(x);
}
catch(InvalidOperationException x)
{
  throw new PortWriteFailedException(x);
}

try
{
  int numRead = port.Read(response, 0,
    response.Length);
  . . .
}
catch(InvalidOperationException x)
{
  throw new PortReadFailedException(x);
}
```

Listing 4

expressiveness. In this case, making the code adequately robust results in a serious detraction from both. Thankfully, we can rely on old reliable '*another level of indirection*' by wrapping all the filth in a class, although we cannot hide from the burden of creating appropriate exception types, nor hide our client code completely from the burden of understanding and coupling to them. All the details of which I now cunningly leave until next time. And now for something slightly different ...

Quench.NET

Quench is an open-source library designed to:

1. Facilitate diagnosis and correction of badly-written application code that inappropriately quenches exceptions; and
2. Provides assistance in the use (and correction of that use) of badly-designed standard and third-party components that indicate failure, via thrown exceptions, without providing adequate information to delineate unambiguously between practically-unrecoverable and recoverable conditions

In both regards, **Quench** facilitates post-hoc discovery and adjustment of application-behaviour, while ensuring that 'safe' defaults are applied, at varying levels of precision.

Quench.NET is the first (and currently only) application of the **Quench** design principle; others will follow in due course.

There be dragons!

Consider the (C#) code fragments in Listings 5–8.I find it deeply concerning to see such code in production software. Over the last couple of years I've had occasion to work with codebases containing literally thousands of inappropriate exception quenches; indeed, the extent of such constructs in one codebase meant that a case-by-case remediation was

```
try
{
. . . // something important
}
catch(Exception /* x */)
{
  return someValue;
}
```

Listing 5

quite impractical. (I find it even more disheartening to see code such as this in strongly-selling and widely-recommended text books – I've encountered the second in one such book I read last year. There is a considerable challenge in writing worthwhile material about programming because publishers – and readers, according to publishers – want only pithy tomes that can fit in a pocket and be read in a day. As a consequence, many books show simplistic views of real programs (and program fragments) that may not reflect fairly their author's practice in order to focus on their subject and to present digestible quanta of material to the reader. As I have already acknowledged in [QM#5], this is a hard conflict to redress satisfactorily: certainly, I am not sure that I have not erred previously in this way myself. Nonetheless, now having realised the full import and complexity of failure-handling, I can't forget it, and I can't forgive books and articles that mislead, since I meet far too many programmers who have been misled.)

In the first case (Listing 5), the user intends to try an operation that may fail (and will indicate its failure by throwing an exception), and to provide some reasonable default instead if it does so. This is a perfectly reasonable intent, just realised badly. The problem is, catching **`Exception`** (rather than the appropriate precise expected exception type) is far too broad: every exception in the .NET ecology derives from **`Exception`**, and this code will quench (almost) all of them, including those that are practically-unrecoverable (such as **`System.OutOfMemoryException`**). **Dragon!**

The way to do this properly is as shown in Listing 9: it's hardly any more effort to catch only the exception representing the failure we want to intercept, rather than (almost) all possible failures.(Note: prior to .NET 2 this **`Parse()&catch`** was the way to try-to-parse a number (or date, or IP address, or ...) from a string. Thankfully, this ugly and inefficient technique was obviated with the introduction of the **`TryParse()`** methods, which return **`true`** or **`false`**, and do not need to throw **`System.FormatException`**.)

The second case (Listing 6) also represents good intentions, and has a veneer of robustness insofar as it issues a diagnostic log statement. Alas, this is specious comfort. First, diagnostic log statements are subject to the *principle* of removability [QM#6], so provision of a diagnostic log statement alone is an invitation to do nothing. Rather, the (removable) diagnostic log statement should be associated with additional (non-removable) action, whether that be to set a flag, throw a different exception, return, or whatever. Second, we still have the huge but subtle issue that we're catching *everything*, including those things that should denote practically-unrecoverable conditions.

Furthermore, the text book to which I've alluded above has a construct like this – albeit that it's notionally a *contingent* report, in the form of **`Console.WriteLine(x);`** – at the outermost scope in **`Main()`**, so it fails the requirement to indicate failure to the operating environment

```
try
{
 . . . // something important
}
catch(Exception x)
{
  LogException(x);
}
```

Listing 6

```
try
{
. . . // something important
}
catch(Exception x)
{}
```

Listing 7

```
try
{
. . . // something important
}
catch
{}
```

Listing 8

[QM#6]: any exception is caught and yet the program returns 0, indicating successful execution, to the operating environment. **Dragon!**

The third case is not even defensible from the position of good intentions gone bad. This is flat-out, unequivocal, inexcusable, malpractice. If you write code such as this, you don't deserve to earn a crust as a programmer. If you encounter code such as this and don't raise the alarm to your team lead, manager, head of IT..., then you're being incredibly reckless and risking having to deal with system failures that you may be literally clueless to diagnose. The problem is a distilled version of the slip we've seen in the first two cases: when you write code such as this you are asserting 'I know everything that can possibly happen and I deem it all to be of no consequence'. Apart from the most trivial cases, I doubt anyone can stand behind either half of that assertion. **Dragon!**

The fourth case (Listing 8) is the same as the third (Listing 7) – just a bit of syntactic sugar(!) provided by the C# language, to avoid the unused reference warning that would result from compiling the code from Listing 7 at warning level 3 or 4. Here, the language designers have gone out of their way to ease the application of a total anti-pattern. *Honestly!* (More a case of **Bats in the Belfry** than **Dragon!**)

Insufficient information to determine recoverability

As I mentioned earlier, it is conceivable that other exception types may be thrown in circumstances not yet considered, indeed, in circumstances never encountered before the system goes into production. As we've already seen with the ambiguity in the serial port `Write()` operation, sometimes exceptions may be thrown that should be quenched and handled in ways that are already established for other types.

But a key plank of the exception paradigm is that if you don't know of/about an exception type, you cannot be written to expect it and should instead allow it to percolate up to a layer that can handle it, albeit that that handling may mean termination.

How do we deal with this situation?

Quench(.NET) to the rescue

In the first situation – the carelessly high-level `Exception` quenching – the answer was to rewrite all the offensive constructs in terms of

```
int i;
try
{
  i = Int32.Parse(s);
}
catch(System.FormatException)
{
  i = -1;
}
```

Listing 9

```
try
{
  . . . // something important
}
catch(Exception x)
{
  LogException(x);
  if(Quench.Deems.CaughtException.MustBeRethrown
      (x))
  {
    throw;
  }
}
```

Listing 10

Quench.NET + diagnostic logging facilities + a `throw` statement, as shown in Listing 10, which illustrates Quench's (attempt at a) fluent API. Due to the sheer number of cases, and the fact that most of them followed just a few simple forms, more than 90% of the cases were effected automatically by a custom-written Ruby script; the remainder by hand and/or IDE macro.

Changing the behaviour of a large commercial production system with a mountain of such technical debt, even if the changes are to stop doing the wrong thing (of ignoring important failures), is a sensitive and risky undertaking. Even though theoretically changing things for the best, the complex behaviour profile is something to which the 'organisational phenotype' – the business systems, the admin and support function, the development team, the users, the customer assistance operatives – has become accustomed. Because of this, it was important to allow the system to carry on precisely as is, and to tune its behaviour in a slow, incremental, methodical, and observed way. Quench supports this because its default response to the (fluent API) question `Quench.Deems.CaughtException.MustBeRethrown()` is 'yes' (`true`). This method (and others in the API) are overloaded in order to allow the caller to specify more precisely the catch context, allowing fine-grained tuning of recoverability for exception types and/or catch contexts; I'll provide more details next time.

In the second situation – the use of 'surprising' APIs, usually (in my experience) on operating system façades – the solution looks much the same. The difference is that this is not retrofitted in extremis, but is designed and coded up-front. Listing 11 is an extract of some a daemon control routine from one of our current systems under development. Since I (have learned to) mistrust expectations (and documentation) about what .NET APIs will throw, I pre-empt any possible surprises by using Quench. I want practically-unrecoverable exceptions (such as `OutOfMemoryException`, already acknowledged with its own `catch`) that don't directly pertain to a failure of registration *per se* to be handled at a higher level (and stop the program), and since I do not (yet) know all the exceptions that may emanate from the system registration, I employ Quench to allow me to tune this at runtime (likely via configuration); integration testing (and, unfortunately, actual use) may inform more definitely, in which case the code can be refactored to catch (and rethrow) specific exceptions rather than use Quench.

There's a third, lesser motivation for using Quench, albeit one that, based on my experience, I think is particularly relevant with .NET. Consider the case where we've done the mooted experiential learning in testing/production and now wish to remove Quench, since its use rightly gives us an uneasy feeling that we've somehow done the wrong thing. However, it can be the case that we've identified a large number of such exceptions, and either they do not share a base class that we could catch in their stead, or some of their peer classes are ones that we do not wish to catch. Whatever the reason, we're now left with a large number of catch-handlers, several of which we wish to take the same action, as in Listing 12. In this case, use of Quench may be the lesser of two evils, since the list of which exceptions can be quenched (and, by inference, which others must be rethrown) can be maintained, either in code or in configuration, more flexibly and neatly. This is even more advantageous where we may find

```
private static bool
DoInstallationOperation(
string installationOperationName
, Action<AssemblyInstaller, IDictionary> func)
{
  using(Pantheios.Api.Scope.MethodTrace
    (Severity.Debug))
  {
    try
    {
      AssemblyInstaller installer =
        new AssemblyInstaller
        (Assembly.GetEntryAssembly(),
        new string[0]);
      IDictionary state = new Hashtable();
      func(installer, state);
      installer.Commit(state);
      Pantheios.Api.Flog(RegistrationLog,
        Severity.Notice,
        "{0} service installed successfully",
        Program.Constants.ProcessIdentity);
      Console.Out.WriteLine("{0} service
        installed successfully",
        Program.Constants.ProcessIdentity);

      return true;
    }
    catch(OutOfMemoryException)
    {
      throw;
    }
    catch(Exception x)
    {
      Pantheios.Api.Log(Severity.Alert,
        "Could not ", installationOperationName,
        " service ",
        Program.Constants.ProcessIdentity,
        ": ", Pantheios.Api.Insert.Exception(x));
      if(Quench.Deems.CaughtException
        .MustBeRethrown(x, typeof(Program)))
      {
        throw;
      }

      Console.Error.WriteLine("Could not {2}
        service {0}: {1}",
        Program.Constants.ProcessIdentity,
        Pantheios.Api.Insert.Exception(x),
        installationOperationName);
    }
  }

  return false;
}
```

Listing 11

ourselves having the same list of quench-vs.-throw rules in several similar contexts.

In whichever case, Quench is not a library to be applied lightly, and it should never be used as a licence to slack off on thinking, reading (documentation), or testing. But, when you have absolutely, positively got to handle every exception in the room, accept no substitutes! ■

In the next issue

I'm somewhat gun-shy about making predictions of when, but I do feel certain that the next instalment, when it comes, will consider more definitively the kinds of information that exceptions should contain, including how to (re)define exception hierarchies that offer rich

```
public static void Blah1()
{
  Exception y = null;

  try
  {
    . . . // complex operation,
          //can throw many exceptions
  }
  catch(OutOfMemoryException)
  {
    throw;
  }
  catch(SomeLeafException x)
  {
    y = x;
  }
  catch(AnotherLeafException x)
  {
    throw;
  }
  catch(SomeParentException x)
  {
    y = x;
  }
  catch(SomeEntirelySeparateException x)
  {
    y = x;
  }
  catch(AnotherEntirelySeparateException x)
  {
    y = x;
  }
  catch(Exception)
  {
    throw;
  }
  System.Console.WriteLine("{0}: ", y);
}
```

Listing 12

information on which one can base solid recovery decisions, along with some/all of the following:

- the design principles, implementation, and customisation and use of Quench.NET. In the meantime, please check it out (at http://www.libquench.org/);
- details of the serial port abstraction, and the supporting exception hierarchy; and
- maybe the new **STLSoft** C++ exception hierarchy (if I get time between all my C# coding).

References

[ENUMS] Enumerating Experiences, Matthew Wilson, *CVu*, September 2011

[QM#5] Quality Matters 5: Exceptions: The Worst Form of 'Error' Handling, Apart from all the Others, Matthew Wilson, *Overload* 98, August 2010

[QM#6] Quality Matters 6: Exceptions for Practically-Unrecoverable Conditions, Matthew Wilson, *Overload* 99, October 2010

Static – A Force for Good and Evil

We've all learnt to avoid the use of the static keyword. Chris Oldwood questions this wisdom.

I've noticed a trend among C# programmers which is to avoid the use of the `static` keyword. It seems I'm not the only one who's noticed this either [Twitter]. It's not inherently limited to C# programmers as C++ can be written in a similar manner, but the terminology bias (functions vs. methods) and its clearer multi-paradigm stance means it's probably less susceptible.

There is a perception that 'static' things – data and methods – are bad. In the wrong hands that can be true, but by throwing the proverbial baby out with the bathwater we have closed the door on embracing some of the goodness that functional-style programming brings.

This article attempts to dispel the myths by illustrating *which* uses of `static` are bad and which are actually beneficial.

Shared mutable state

My guess is that the `static` keyword has got a bad rap because of past transgressions caused by functions that were designed decades ago in a time when re-entrancy and multi-threading were something only specialist programmers had to worry about. Yes `strtok()` I'm looking at you.

This old C function which is used to tokenise a string has some serious side-effects. Behind the scenes it keeps track of the string being tokenised (which it also modifies) so that you can keep calling it without supplying the original input string when fetching the next token:

```
char input[] = "unit test ";
char separators[] = " ";
assert(strcmp(strtok(input, separators),
       "unit") == 0);
assert(strcmp(strtok(NULL, separators),
       "test") == 0);
```

In a single-threaded environment you have to be careful not to 'nest' use of it (e.g. tokenise a token), and in a multi-threaded environment this kind of behaviour is a disaster waiting to happen. Fortunately many C implementations managed to avoid ruining a programmer's day due to spurious errors by utilising thread-local storage, but this was a courtesy and not standards-defined behaviour.

The anti-pattern, for want of a better term, which can lead to this kind of sorry state of affairs, is to take a simple function that only depends on its inputs and then find you need to add new behaviour without changing its interface. In a waterfall-esque development process the new behaviour could be the need to cache results, and the inability to change the interface might come from discovering this very late during The Testing Phase. Of course adding a cache and then accidentally making it non-thread-safe is only going to exacerbate your woes at this point of the cycle. More likely it doesn't need to be thread-safe *initially* but does later; only no one notices it's not. The C# example in Listing 1 shows how easy it can be to naively add a cache to an existing class.

```
public static class ThingyProcessor
{
  public static int CalculateThing(int input)
  {
    int output;
    if (Cache.TryGetValue(input, out output))
      return output;

    // Long calculation...

    Cache.Add(input, output);
    return output;
  }
  // Non-thread-safe collection
  private static Dictionary<int, int> Cache =
                 new Dictionary<int, int>();
}
```

Listing 1

Sharing mutable state via global variables (`public static` properties) is definitely a very bad smell and has been for many years, but also sharing it implicitly across threads can be dangerous too. It's not just a matter of ensuring our own types are safe, it's the whole object graph, so any 3rd party collection types have to be checked too.

Although I've focused on complex types above, it should be noted that primitive values are not immune from this problem either. If anything they are likely to 'appear to work' more than a complex type due to their small footprint. Use of `volatile` and the various `Interlocked` helper methods are required to keep them behaving properly.

Sometimes sharing mutable state is a necessary evil but there should be precious few times when we need to enter those murky waters. If possible we should look to change the interface or find some other design to make the problem disappear altogether and keep the code simple.

Before moving on let's just go back to the procedural world of C to look at how we might tackle this problematic function. Putting aside for the moment the fact that `strtok()` is a published function specified by a standards process, we could avoid its state problem by changing the interface to allow the state to passed back in by the caller. Also we'd tackle the mutation of the input string to restrict the side-effects to just the state object. (See Listing 2.)

There is still more that could be done to improve matters, such as using two separate functions (e.g. `firsttok()`/`nexttok()`). But this is C and combining state and functions into a more cohesive package is exactly what object-orientation allows us to do more cleanly in languages like C++ and C#. Hence in OO you might choose to present a `strtok`-style class in C# like Listing 3.

Chris Oldwood is a freelance developer who started out as a bedroom coder in the 80s, writing assembler on 8-bit micros. These days it's C++ and C# on Windows in big plush corporate offices. He is the commentator for the Godmanchester Gala Day Duck Race and can be contacted via gort@cix.co.uk or @chrisoldwood

```
typedef struct strtok
{
  const char* string;
  const char* separators;
  const char* tokenBegin;
  const char* tokenEnd;
} strtok_state_t;

const char* input = "unit test ";
const char* separators = " ";
strtok_state_t state;

strtok(input, separators, &state);
assert(strncmp(state.tokenBegin, "unit",
state.tokenEnd - state.tokenBegin) == 0);

strtok(NULL, NULL, &state);
assert(strncmp(state.tokenBegin, "test",
state.tokenEnd - state.tokenBegin) == 0);
```

Listing 2

Shared immutable state

When talking about the pitfalls of 'shared state' it's important to qualify what sort of state you're talking about. As we just discussed, shared *mutable* state can be dangerous when done badly. In contrast shared *immutable* state is much safer; at least once the potentially tricky initialisation phase is complete. Once we have a read-only data structure it can be used concurrently without the need for any kind of locking. Even if it can be referenced globally, which may still be another smell; it cannot be changed behind our backs.

For example, imagine you're writing a simple XML parser in C#. To handle the translation of entity references, such as **`&`** to their equivalents you might decide to use a lookup table. This table will likely be immutable and so it requires no additional locking in the event that two threads attempt to parse separate XML documents concurrently. (See Listing 4.)

The initialisation issue is handled for us by the C# language which guarantees that type constructors are thread-safe. That said we need to be especially careful inside a type constructor because if it throws things start going horribly wrong as the type can't be loaded.

Static classes

C#, unlike C++, does not allow methods to exist outside of classes (often called free functions in C++). Consequently you are forced into defining

```
public class StringTokeniser
{
  public StringTokeniser(string input,
    string separators)
  {
    // Remember inputs
  }

  public string NextToken()
  {
    // Find next separator or the string end by
    // searching onwards from the last 'position'.
  }

  private readonly string _input;
  private readonly string _separators;
  private int _position;
}
```

Listing 3

```
public class XmlParser
{
  private static string DecodeEntity(string
entity)
  {
    string output;

    if (EntityTable.TryGetValue(entity,
        out output))
      return output;
    return entity;
  }

  private static Dictionary<string, string>
    EntityTable = new Dictionary<string, string>
  {
    { "&amp;", "&" },
    { "&gt;",  ">" },
    { "&lt;",  "<" },
  };
}
```

Listing 4

a class even when all you want to write is a simple function. I suspect this has an undesirable side-effect on C# programmers because I've seen them create classes that hold no state, or only hold compile-time immutable state (i.e. constants), e.g.:

```
public class ConfigurationSettings
{
  public string DatabaseName { get
    { return ". . . "; } }
}
```

It's not always as obvious as this and it might be returned as a property of another class. These extra levels of indirection make it harder for a static code analysis tool to spot it and suggest a refactoring.

```
public class Configuration
{
  public ConfigurationSettings Settings { get
    { return _settings; }}
  private readonly
    ConfigurationSettings _settings =
    new ConfigurationSettings();
}
```

In a managed environment like C#, classes such as the **`ConfigurationSettings`** class above are quite literally garbage – the objects just get created and then destroyed again and their behaviour can be determined entirely at compile-time. As with free functions, constants in C# need to be defined as part of a class too:

```
public static class ConfigurationSettings
{
  public const string DatabaseName = ". . . ";
}
```

The C# answer to classes which shouldn't be instantiated is to declare them 'static'. In essence the class is now acting as merely a namespace, albeit one that you can't elide with a **`using`** declaration at the top.

The canonical example in C# for a static class of 'pure' functions (deterministic functions that only depend on their inputs and have no side-effects) is probably the **`Math`** class which plays hosts to fundamentals like **`Abs()`** and **`Min()`**.

```
public static class Math
{
  public static int Abs (int value)
  {
    return (value < 0) ? -value : value;
  }
}
```

Static functions

Right back at the beginning I suggested one reason why there might be a fear of static functions is because of where their implementation could end up. I'd also suggest that programmers find it easier to pass parameters to functions by making them *implicit*, i.e. through class members accessible via `this`.

Back in the 1980s, Meilir Page-Jones wrote a book called *The Practical Guide to Structured System Design*. He goes into detail about the various types of coupling we might see in code, with each category being viewed as a less desirable form from a maintenance perspective. Whilst the most serious forms of coupling should be avoided, Page-Jones suggests that the weaker forms can be used effectively in the right hands, but also have the *potential* to cause grief in the wrong ones.

At the farthest end of the spectrum we have Content Coupling which is of little concern in today's languages. Back in the days of assembler programming you could jump from the middle of one 'function' right into the middle of another, meaning you couldn't ever be sure where you'd come from. Next up is Common Coupling, i.e. global variables. As the name implies they have the ability to affect any and every part of the code-base in unanticipated ways.

Then we come to Control Coupling. This is where one function passes some kind of flag or signal to tell another how to behave. Depending on the direction of the signal either a child is telling its parent how to behave or the parent might know too much about the child's implementation, either way it's a symptom of low cohesion.

Moving onwards we come to Stamp Coupling. This oddly named formed of coupling is about passing excessive input to functions that don't need it. For example, if you had a function that formatted a customer's full name from their first and last names, you should consider only passing those two arguments, not an entire Customer record. By passing the entire type you make the function (appear to be) dependent on attributes it doesn't use.

Finally we reach Data Coupling which is analogous to a 'pure' function. What Page-Jones tells us is that the easiest code to reason about is this style of function which, as mentioned earlier, has a deterministic output solely based on its direct inputs with no side-effects. In essence, given that *some* form of coupling is a necessity to do anything useful, then Data Coupling is the most preferable.

Member coupling

His book was published in a time before Object-Orientated Programming was A Big Thing. He is also concerned more with inter-module coupling rather than intra-module coupling, such as between methods of the same class. As the size of a class grows it becomes more common to rely further and further on data being passed between methods via its own state, i.e. its members. I believe there is a form of Stamp Coupling going on here as any method might use its input arguments plus any aspect of the object's current state or per-class state, and so it is impossible to know what that is without looking at the implementation of an instance method. And that is what coupling is all about – being able to reason about the knock-on effects of a change to other parts of the code.

The over reliance of implied state makes it harder to refactor code later because pulling that state out to another data structure may require lots of unexpected fixing up of other methods. I've found that taking Page-Jones's advice to favour Data Coupling right into the heart of classes has made code easier to read because simple methods start looking like simple black boxes again.

As a simple example consider the class-based OO version of the `strtok()` function I mentioned earlier. In the implementation, when it comes to finding the next token we could rely solely on the implied state held in the member variables and code it up as one method (see Listing 5).

One alternative would be to hand-off the finding off the end of the token to a separate little method that is only dependent on its inputs (Listing 6).

Although the first implementation of `NextToken()` is quite small there is perhaps a temptation to put a comment above the bit of code that finds the end of the token because it's not immediately apparent due to the overloaded use of the `_position` member (initial start of the next token and then the end of next token). Whenever I find myself wanting to write a comment I consider that to be a sign I should use the Extract Method or Introduce Explaining Variable [Fowler] refactorings instead.

```
public class StringTokeniser
{
  . . .
  public string NextToken()
  {
    if (_position == _input.Length)
      return null;
    var start = _position + 1;
    _position =
      _input.IndexOf(_separators, start);
    if (_position == -1)
      _position = _input.Length;
    return _input.Substring(start,
     _position - start);
  }
  private readonly string _input;
  private readonly string _separators;
  private int _position = -1;
}
```

Listing 5

```
public class StringTokeniser
{
  . . .
  public string NextToken()
  {
    if (_position == _input.Length)
      return null;
    var start = _position + 1;
    var end = FindTokenEnd(_input, _separators,
      start);
    var token = _input.Substring(start,
      end - start);
    _position = end;
    return token;
  }
  public static int FindTokenEnd(string input,
    string separators, int start)
  {
    var end = input.IndexOf(separators, start);
      if (end == -1)
        return input.Length;
    return end;
  }
  private string _input;
  private string _separators;
  private int _position = -1;
}
```

Listing 6

There might be a knee-jerk reaction that splitting code up into so many simple methods would create a big hit on performance. It is possible, but then we all know that premature optimisation is a dangerous pastime. The JIT compiler in .Net and modern C++ compilers can do a pretty good job these days of inlining methods so you'll probably not notice it in the vast majority of your code.

Exception safety

Whilst it's highly unlikely that any client code would attempt to recover directly from an OutOfMemory exception thrown from our `StringTokeniser` class, it is a library function and they often get used

```
public class ProcessManager
{
  . . .
  public void ExecuteJob(Job job)
  {
    if (_process == null)
    {
      _process = new Process();
      _process.Start(_applicationName);
    }
    . . .
    _process.Execute(job);
  }
  . . .
  private readonly string _applicationName;
  private Process _process;
}
```

Listing 7

in mysterious ways. Writing exception safe code is hard, especially when it's so easy to mutate state at an unsafe moment.

A common example I've seen of this is when two-phase construction is used and the second phase throws an exception, leaving a member mutated by accident (see Listing 7).

Here, if `ExecuteJob()` throws when the `Start()` method is called the `_process` member will be left pointing to a partially initialised object. When the second call to `ExecuteJob()` comes in it will assume the `_process` member is fully initialised and will try and to use it.

The general pattern for writing exception safe code is to perform all code that might throw off to the side and then commit the changes locally with non-throwing operations. In this example we could have written it like Listing 8.

Internal factory methods are a good fit for being static because object creation and initialisation is often full of code likely to throw that you might want to keep at arms length until you know you're dealing with fully constructed objects. By factoring the creation out into a static method you also convey to both the compiler and the reader that they shouldn't be messing with any of `this` object's state at that point of its lifecycle (Listing 9).

Extension methods

Whilst C# might not support 'free functions', it does have Extension Methods and these often embody the practice of writing small independent methods. They are members of a static class and are themselves declared static, despite the fact that they appear to be called as instance methods. If you ever wished that the C# String class had an instance method that could tell you whether a string was empty or just contained white-space, you can make it happen yourself (see Listing 10).

This is often how I find extension methods come about. Initially they start as a simple static method in a class that looks suspiciously as though the first argument really wants to be `this`. Once reuse rears its head, it's a

```
public void ExecuteJob(Job job)
{
  if (_process == null)
  {
    var process = new Process();
    process.Start(_applicationName);

    _process = process;
  }
  . . .
  _process.Execute(job);
}
```

Listing 8

```
public class ProcessManager
{
  public void ExecuteJob(Job job)
  {
    if (_process == null)
      _process = CreateProcess(_applicationName);
    . . .
    _process.Execute(job);
  }

  private static Process
    CreateProcess(string applicationName)
  {
    var process = new Process();
    process.Start(applicationName);
    return process;
  }
  . . .
  private Process _process;
}
```

Listing 9

simple step to factor it out into a common extension method. Alternatively it could be pulled out as a formal extension method, but left defined inside a private static class of the current consumer to avoid publishing it formally as that comes with the possible burden of needing to write separate unit tests.

Building classes from static methods

The Object-Orientated paradigm is good for creating types that represent things, but when it comes to algorithms and processes it can start to get ugly. Take the process of parsing a string of XML into a DOM for example. Whilst the input string can be an object, and the output is a tree of objects, the algorithm used to process the characters in the string and create the tree of objects feels much less object-like. It feels to me more like a function that transforms one to the other. Yes, there will be some temporal state involved during the processing, but by-and-large the decomposition of the problem has more of a focus on functions.

If I was using Test-Driven Development to tackle a problem like this my initial tests would very likely start out with just a simple function (Listing 11). You can argue about whether it should be a member of a class called

```
public static class StringExtensions
{
  public static bool IsBlank(this string value)
  {
    return (value.Trim().Length == 0);
  }
}

public static class Program
{
  public static int Main(string[] args)
  {
    . . .
    string connectionString =
      configuration.ConnectionString;

    if (!connectionString.IsBlank())
      connection =
      OpenConnection(connectionString);
    . . .
  }
}
```

Listing 10

```
[Test]
public void when_xml_is_empty_then_dom_is_empty()
{
  const string emptyXml = "";
  var document =
    XmlParser.ParseDocument(emptyXml);
  Assert.That(document, Is.Not.Null);
}
```

Listing 11

```
[Test]
public void when_xml_is_empty_then_dom_is_empty()
{
  const string emptyXml = "";
  var parser = new XmlParser();
  var document = parser.ParseDocument(emptyXml);
  Assert.That(document, Is.Not.Null);
}
```

Listing 12

`XmlDocument`, or `XmlReader`, etc. but either way I wouldn't start out expecting to create a class like Listing 12.

Anyone who uses FizzBuzz [FizzBuzz] or the Roman Numerals kata in their interview process to separate the 'wheat from the chaff' will probably see this kind of thing. It's not wrong, *per se*, but it can lead to the kind of 'empty' classes described above.

From a Design Pattern's perspective what my eventual function will become is akin to a façade over a bunch of other functions. As the number of tests grow, so will the number of internal functions. Internal refactoring will start to push some of those out into separate (internal) classes which in turn will likely receive their own more focused unit tests. The handling of XML entity references earlier was very simplistic, just a lookup table, but as more scenarios are discovered so the complexity of that aspect of the implementation will likely increase.

The state required during parsing is entirely transient from the perspective of the caller. It could be held inside an instance of the `XmlParser` class, where the public class gets a private constructor because it just becomes an implementation detail of the static `ParseDocument()` method. But why even expose that, why not create an internal class, say, `XmlParserImpl` and treat `ParseDocument()` as a sort of top-level factory method? (See Listing 13.)

If the implementation class is internal then the entire type is encapsulated and so we could hold the state as a Dumb Data Object [DDO] and access it in our static methods via public fields (so long as we promise never to expose it). Then again we could hold the state entirely on the stack by passing it as parameters to recursion functions.

```
internal class XmlParserImpl
{
  . . .
}
public static class XmlParser
{
  public static Dom ParseDocument(string xml)
  {
    var parser = new XmlParserImpl(xml);
    return parser.ParseDocument();
  }
}
```

Listing 13

Methods on immutable types

Jon Skeet raised a question at the Norfolk Developers Conference [Norfolk] about how to make methods on immutable types more intention revealing. His canonical example in C# involves the `DateTime` class like this:

```
DateTime date = DateTime.Today;
date.AddDays(1);
```

The method name `AddDays` suggests that it will add 1 day to `date` and give us the date for tomorrow. Only it won't. It will create a new `DateTime` value based on `date` (with an offset) which will subsequently be thrown away by the caller. It's an easy mistake to make and one of the reasons why writing tests is such a worthy pursuit. The example should have been:

```
DateTime date = DateTime.Today;
DateTime tomorrow = date.AddDays(1);
```

Jon went on to question whether there is a way to name methods to make this pattern (returning a new value instead of mutating the existing one) more revealing. He proposed this for the example above:

```
DateTime tomorrow = date.PlusDays(1);
```

It is an improvement, but I would posit that it's just too subtle a change in language to really make a difference. Part of the problem is that mutability is the default position in C#; for example the object initializer syntactic sugar relies on the class having writable properties.

My own stance is that once again we can draw from the functional side and use static methods (aka functions) to more clearly suggest that a new value will be created from an existing one and some adjustment:

```
DateTime tomorrow = Date.AddDays(date, 1);
```

If you started making the same mistake above with this style, would it be any more obvious?

```
Date.AddDays(date, 1);
```

At least this way you start by invoking a static method and so that should tell you something extra that you don't get by invoking an instance method.

Summary

The `static` keyword is in need of a public relations exercise in C# to try and overcome prejudices caused by misunderstanding its role. C# might have started out with a heavy bias towards the object-orientated paradigm but over the years its audience and the language have tried to embrace a multi-paradigm world. This means a stronger focus on immutability and the use of functions instead of objects and mutable state, at least for those problems where it's beneficial.

The natural outcome of this is code that is easier to reason about and inherently thread-safe. Whilst another language such as F# might be a better tool for the job by removing some of the ceremony, there is no reason why you cannot adopt some of their practices to improve a C# codebase. ■

Acknowledgements

A debt of gratitude is owed to Ric Parkin and Roger Orr for helping me polish this article.

References

[DDO] http://c2.com/cgi/wiki?DumbDataObject

[FizzBuzz] http://c2.com/cgi/wiki?FizzBuzzTest

[Fowler] http://martinfowler.com/books/refactoring.html

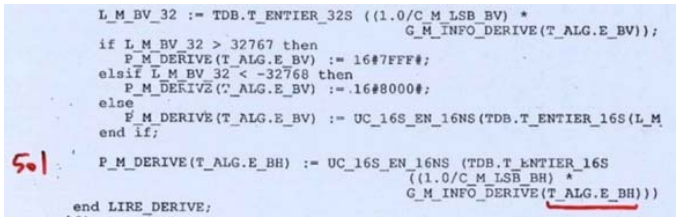
[Norfolk] http://nordevcon.com/

[Twitter] https://twitter.com/codemonkey_uk/statuses/385518295958683649

Search with CppCheck

Finding code of interest is a vital skill but our tools are often too simple. Martin Moene tries something better.

Writing software isn't what it used to be. 'But I enjoy it more than ever', you say. Yes, indeed. I mean, we write differently, our writing style has changed. And sometimes we may like to rewrite history a little and change some old-fashioned code to something contemporary.



```
L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
                               G_M_INFO_DERIVE(T_ALG.E_BV));
if L_M_BV_32 > 32767 then
   P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
   P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M
end if;

P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                               ((1.0/C_M_LSB_BH) *
                               G_M_INFO_DERIVE(T_ALG.E_BH)))
end LIRE_DERIVE;
```

ADA source code excerpt that led to the rapid unscheduled disassembly of Ariane 5, Flight 501 [Ariane-5].

Figure 1

Take for example software I'm working on and that contains code not unlike the Ariane 5 code in Figure 1 in several places. (The definition of **`uiMinValue`** and **`uiMaxValue`** is not shown here.)

```
void foo( const int uiValue )
{
  int uiTrueValue = uiValue;
  if ( uiTrueValue < uiMinValue )
  {
    uiTrueValue = uiMinValue;
  }
  else if ( value > uiMaxValue )
  {
    uiTrueValue = uiMaxValue;
  }
  //use uiTrueValue...
}
```

Use a more intent-revealing name and change the formatting and it's easier to see what the code is about, as similarities and differences in the code fragment now stand out.

```
void foo( const int value )
{
  int clampedValue = value;
  if      ( clampedValue < minValue )
    clampedValue = minValue;
  else if ( clampedValue > maxValue )
    clampedValue = maxValue;
  // use clampedValue...
}
```

Although I like this kind of code formatting, it touches on the *unsustainable* [Henney11]. It's also still rather long winded and **`clampedValue`** is mutable while there's no reason why it should be. So, let's change the code further to use a single assignment.

```
void foo( const int value )
{
  using namespace std;
  const int clampedValue =
    min( max( value, minValue ),maxValue );
  // use clampedValue ...
}
```

A nice C++ idiom. Or so I thought. Until I looked at such code with a domain expert. I had to explain that this code clamps the given value between two extremes. And I agree: the code tells what it's intended for in a rather obscure way. Why not just say **`clamp`** or **`limit`**?

```
void foo( const int value )
{
  const int clampedValue =
    clamp( value, minValue, maxValue );
  // use clampedValue ...
}
```

It turns out that **`clamp`** is one of the algorithms in library **`Boost.Algoritms`** [Boost]. Microsoft's C++ AMP library for parallelism also provides a **`clamp`** function [AMP]. Several interesting suggestions for such a function were made on mailing list accu-general [Alternative].

CppCheck rules

Knowing what we have and what we want to change it to, the next question is: how do we find all occurrences we want to change?

Of course we can try and use grep or the search function of our IDE, but it may prove difficult to specify a search that allows for sufficient variation and that may span multiple lines.

Now I had been using CppCheck occasionally to assess code from the CodeBlocks IDE [CppCheck]. When I noticed a new version was available I glanced over the manual and noticed CppCheck allows you to specify simple rules via an XML format specification [Manual].

```
<?xml version="1.0"?>
<rule>
  <tokenlist>LIST</tokenlist>
  <pattern>PATTERN</pattern>
  <message>
    <id>ID</id>
    <severity>SEVERITY</severity>
    <summary>SUMMARY</summary>
  </message>
</rule>
```

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

Without further parameters, CppCheck only shows error messages and matches to our own rules

With such a rule, CppCheck can search for a pattern in the code that may span multiple lines. PATTERN, a Perl-compatible regular expression [PCRE] controls which code patterns will match the search. For our purpose, **LIST** is specified as simple and can be omitted. Examples for **ID**, **SEVERITY** and **SUMMARY** of an existing message are: **variableScope**, **style** and **"The scope of the variable 'var' can be reduced"**.

Please don't also fall into the trap of thinking your PCRE knowledge suffices to proceed and specify a search pattern. Instead start with reading *Writing Cppcheck rules* [Rules].

CppCheck search pattern

As a first step to compose a regular expression for the pattern you are interested in, run CppCheck with rule **.+** on a code excerpt that contains the pattern.

```
prompt> cppcheck.exe" --rule=".+" file-with-code-
pattern.cpp
```

For the first example above, this gives:

```
[example1.cpp:1]: (style) found \
' int foo ( const int uiValue ) { \
int uiTrueValue ; uiTrueValue = uiValue ; \
if ( uiValue < uiMinValue ) { uiTrueValue =
  uiMinValue ; } \
else { if ( uiMaxValue < uiTrueValue ) {
  uiTrueValue = uiMaxValue ; } } \
return uiTrueValue ; }'
```

The multiple-line source code is parsed, simplified and presented as a single-line token stream. Note that the second comparison is changed from **>** to **<** and has its left- and right-hand sides swapped. If CppCheck discovers that **uiTrueValue** isn't used further on, it removes the assignments from the **if-else** block. For more information on the transformations CppCheck applies, see [Simplification].

To find the code we're looking for, we may search for the following token stream.

```
if ( ... < ... ) { ... = ... ; } else {
  if ( ... < ... ) { ... = ... ; }
```

Here we choose to match the **if-else-if** language construct and comparison operator. We accept any operand as indicated by the ellipses, and require the presence of an assignment in the **if** blocks of the program.

This token stream can be matched by the following regular expression.

```
if \( \w+ \x3c \w+ \) { \w+ = \w+ ; }
else { if \( \w+ \x3c \w+ \) { \w+ = \w+ ; } }
```

We use **\x3c** for the comparison, as specifying **<** or **\<** does not work, due to the embedding in XML. Change **\x3c** to **\x3c=?** if you want to match both less-than and less-equal. If you want to match no-matter-what code instead of the assignment, you can use **{ .*? }** or **{ [^}]*? }** for the block. Be careful though to not specify a pattern that is too greedy. Apparently leaving part of the pattern unused when all input has been consumed does not prevent declaring victory. You'll get a single match at best.

To allow both **if...if** and **if...else if**, specify **(else_{_)?if** for the second **if** in the pattern and omit the terminating **_}** from the pattern. Take note of the space following the opening brace in the **(else_{_)?** part. (The underscore denotes a space.)

Now let's apply CppCheck with a rule for this pattern to a file that contains several variations of **if-else** constructs (available from [GitHub]).

```
prompt> cppcheck --rule-file="my-rule.xml"
sample.cpp
```

This gives:

```
Checking sample.cpp...
[sample.cpp:118]: (error) Address of an auto-
variable returned.
[sample.cpp:5]: (style) Consider replacing if-else-
if with clamp(value, minval, maxval).
... 6 more similar messages
```

Without further parameters, CppCheck only shows error messages and matches to our own rules. You can use option **--enable=...** to enable more kinds of checks from CppCheck itself, such as **style** and **portability** warnings [Wiki]. With option **--template=...** you can get various output formats, e.g. Visual Studio compatible output (**vs**) and GNUC compatible output (**gcc**). Run **cppcheck -h** to see the tool's complete command line usage.

Results

In the circa 400k line codebase at hand, CppCheck found 15 occurrences with the pattern that matches both **if-if** and **if-else-if** and both less-than and less-equal comparisons. Of these, 7 represent code we are interested in and 8 are false positives. Six false positives are due to two long **if-else** chains in a single file and the other 2 are due to the same code of a duplicated file.

Running a check with a looser search pattern for the code in the **if** blocks (**{ .*? }**) gave 31 occurrences but didn't reveal any missed opportunities.

To get rid of the false positives due to longer **if-else-if** chains, we may extend our rule with look-ahead and look-behind assertions [PCRE-man]. A positive look-ahead assertion is specified with **(?=...)**, a negative assertion as **(?!...)**. Look-behind assertions are **(?<=...)** for positive and **(?<!...)** for negative assertions.

To not match **if** when preceded by **else_{_** we can specify a negative look-behind assertion like:

```
(?<!else { )if
```

Unfortunately this pattern contains a **<** which cannot be replaced with **\x3**. To protect it, we enclose the *complete* pattern within **<![CDATA[...]]>** [XML].

```
<pattern><![CDATA[(?<!else { )if \( \w+ <=? \w+ \)
{ [^}]*? } else { if \( \w+ <=? \w+ \) { [^}]*? }
}]]></pattern>
```

This may indicate that a relatively general pattern doesn't necessarily lead to many false positives

Note that we also specified the less-than character in the **`if`** statements as plain **`<`**.

Searching the complete Boost 1.53 source tree with the rule in `my-rule.xml` didn't turn up a single occurrence. This may indicate that a relatively general pattern doesn't necessarily lead to many false positives, or perhaps more likely, that Boost's programming style involves none of these old-fashioned **`if-else-if`** constructs.

Wrapup

This article showed how we can use CppCheck to easily find occurrences of code fragments with a specific structure that may span multiple lines.

To close, a quote from the CppCheck webpage:

> Using a battery of tools is better than using 1 tool. Therefore we recommend that you also use other tools.

For inspiration, see Wikipedia's 'List of tools for static code analysis' [Wikipedia]. ■

Acknowledgements

Many thanks to Ric Parkin and the *Overload* team for their care and feedback. Also thanks to all who made suggestions on the subject of clamping via accu-general [Alternative].

Notes and references

Code for this article is available on [GitHub].

[Alternative] On mailinglist accu-general several people made suggestions about a clamp function.

Phil Nash suggests the following approaches:

- **`clamp( factor ).between( minFactor, maxFactor );`**
- **`clampBetween( factor, minFactor, maxFactor );`**
- **`[Limits clamp: factor between: minFactor and: maxFactor]`** (Objective-C)

Gennaro Prota suggests names **`clamp_to_range`**, **`constrain_to_range`**, **`restrain_to_range`**, **`limit_to_range`**, **`bring_to_range`** and **`to_nearest_in_range`**. He also notes that to make such function **`constexpr`**, it shouldn't be implemented in terms of **`std:min()`** and **`std::max()`**. In turn a reviewer noted that **`std:min()`** and **`std::max()`** are likely to become **`constexpr`** in C++17 (LWG issue 2350, voted into the working paper at the last meeting, in Issaquah).

Initially I used **`limit`** as this word also appears in **`std::numeric_limits`** in the C++ standard library. However Jonathan Wakely argues ... if you propose it [for the standard] I suggest you call it **`clamp`**, I expect that's the most well-known name.

[AMP] Microsoft. C++ Accelerated Massive Parallelism library. http://msdn.microsoft.com/en-us/library/hh265137.aspx

[Ariane-5] Image from presentation 'A Question of Craftsmanship' by Kevlin Henney. InfoQ. 9 March 2014 http://www.infoq.com/presentations/craftsmanship-view. See also 'Cluster (spacecraft)' Wikipedia. http://en.wikipedia.org/wiki/Ariane_5_Flight_501 Accessed on 12 March 2014.

[Boost] The Boost Algoritm library contains a version of **`clamp`** as shown here plus a version that also takes a comparison predicate. http://www.boost.org/libs/algorithm/doc/html/algorithm/Misc.html#the_boost_algorithm_library.Misc.clamp

[CppCheck] CppCheck homepage: http://cppcheck.sourceforge.net/ Cppcheck is a static analysis tool for C/C++ code. Unlike C/C++ compilers and many other analysis tools it does not detect syntax errors in the code. Cppcheck primarily detects the types of bugs that the compilers normally do not detect. The goal is to detect only real errors in the code (i.e. have zero false positives). Cppcheck is supposed to work on any platform that has sufficient cpu and memory and can be used from a GUI, from the command line, or via a plugin.

[GitHub] Code for Search with CppCheck. Martin Moene. 11 March 2014. https://github.com/martinmoene/martin-moene.blogspot.com/tree/master/Search%20with%20CppCheck

[Henney11] Kevlin Henney. 'Sustainable space' *CVu*, 22(6):3, January 2011. Kevlin Henney shares a code layout pattern.

[Manual] CppCheck Manual. http://cppcheck.sourceforge.net/manual.html (HTML format) and http://cppcheck.sourceforge.net/manual.pdf (PDF format)

[PCRE] PCRE – Perl Compatible Regular Expressions. http://www.pcre.org/

[PCRE-man] PCRE man page, section Lookbehind assertions. http://www.pcre.org/pcre.txt

[Rules] Daniel Marjamäki. Writing Cppcheck rules. Part 1 – Getting started (PDF). 2010. http://sourceforge.net/projects/cppcheck/files/Articles/writing-rules-1.pdf/download

[Simplification] Daniel Marjamäki. Writing Cppcheck rules. Part 2 – The Cppcheck data representation (PDF). 2010. http://sourceforge.net/projects/cppcheck/files/Articles/writing-rules-2.pdf/download

[Wiki] CppCheck Wiki (http://sourceforge.net/apps/mediawiki/cppcheck/) Describes checks performed by CppCheck.

[Wikipedia] List of tools for static code analysis: http://en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis Accessed 17 February 2014.

[XML] It took me some time to discover to use the CDATA construct. Once I found out, I couldn't easily find a suggestion of this approach in relation to CppCheck.

I filed this suggestion (https://sourceforge.net/apps/trac/cppcheck/ticket/5551) in the CppCheck issue tracker.

Windows 64-bit Calling Conventions

How the stack works is useful to understanding your programs' behaviour. Roger Orr compares and contrasts.

There are many layers of technology in computing: we even use the term 'technology stack' when trying to name the set of components used in the development of a given application. While it may not be necessary to understand all the layers to make use of them a little comprehension of what's going on can improve our overall grasp of the environment and, sometimes help us to work with, rather than against, the underlying technology.

Besides, it's interesting!

Many of us write programs that run on the Windows operating system and increasingly many of these programs are running as 64-bit applications. While Microsoft have done a fairly good job at hiding the differences between the 32-bit and 64-bit windows environments there are differences and some of the things we may have learnt in the 32-bit world are no longer true, or at least have changed subtly.

In this article I will cover how the calling convention has changed for 64-bit Windows. Note that while this is very *similar* to the 64-bit calling conventions used in other environments, notably Linux, on the same 64-bit hardware I'm not going to specifically address other environments (other than in passing.) I am also only targeting the 'x86-64' architecture (also known as 'AMD 64') and I'm not going to refer to the Intel 'IA-64' architecture. I think Intel have lost that battle.

I will be looking at a small number of assembler instructions; but you shouldn't need to understand much assembler to make sense of the principles of what the code is doing. I am also using the 32-bit calling conventions as something to contrast the 64-bit ones with; but again, I am not assuming that you are already familiar with these.

A simple model of stack frames

The basic principle of a stack frame is that each function call operates against a 'frame' of data held on the stack that includes all the directly visible function arguments and local variables. When a second function is called from the first, the call sets up a new frame further into the stack (confusingly the new frame is sometimes described as 'further up' the stack and sometimes as 'further down').

This design allows for good encapsulation: each function deals with a well-defined set of variables and, in general, you do not need to concern yourself with variables outside the current stack frame in order to fully understand the behaviour and semantics of a specific function. Global variables, pointers and references make things a little more complicated in practice.

The 'base' address of the frame is not necessarily the lowest address in the frame, and so some items in the frame may have a higher address than the frame pointer (+ve offset) and some may have a lower address (-ve offset).

In the 32-bit world the **`esp`** register (the sp stands for 'stack pointer') holds the current value of the stack pointer and the ebp register (the bp stands for 'base pointer') is used **by default** as the pointer to the base address of the stack frame (the use of 'frame pointer optimisation' (FPO) – also called 'frame pointer omission' – can repurpose this register.) Later on we'll see what happens in 64-bits.

Let's consider a simple example function **`foo`**:

```
void foo(int i, char ch)
{
 double k;
 // ...
}
```

Here is how the stack frame might look when compiled as part of a 32-bit program:

	Offset	Size	Contents
High mem	+16		Top of frame
	+13	3 bytes	padding
	+12	1 byte	**`char ch`** (arguments pushed R to L)
	+8	4 bytes	**`int i`**
	+4	4 bytes	return address
`ebp`->	+0	4 bytes	previous frame base
`esp`*->	-8	8 bytes	**`double k`** (and other local variables)
Low mem			

* **`esp`** starts here but takes values lower in memory as the function executes.

Note: the assembler listing output that can be obtained from the MSVC compiler (`/FAsc`) handily displays many of these offsets, for example:

```
_k$ = -8    ; size = 8
_i$ = 8     ; size = 4
_ch$ = 12   ; size = 1
```

This is conceptually quite simple and, at least without optimisation, the actual implementation of the program in terms of the underlying machine code and use of memory may well match this model. This is the model that many programmers are used to and they may even implicitly translate the source code into something like the above memory layout.

The total size of the stack frame is 24 bytes: there are 21 bytes in use (the contiguous range from -8 to +13) but the frame top is rounded up to the next 4 byte boundary.

You can demonstrate the stack frame size in several ways; one way is by calling a function that takes the address of a local variable before and during calling **`foo`** (although note that this simple-minded technique may not work as-is when aggressive optimisation is enabled). For example, see Listings 1.

In the function:

```
check();
foo(12, 'c');
```

Roger Orr has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

Optimisers often re-use memory addresses for various different purposes and may make extensive use of registers

```
void check()
{
  static char *prev = 0;
  char ch(0);
  if (prev)
  {
    printf("Delta: %i\n", (prev - &ch));
  }
  prev = &ch;
}
```

Listing 1

In the caller:

```
void foo(int i, char ch)
{
  check();
  double k;
  // ...
```

The 32-bit stack frame for **foo** may be built up like this when it is called:

In the caller	
`push 99`	set **ch** in what will become **foo**'s frame to 'c'
`push 12`	set up i in foo's frame
`call foo`	enter **foo**, return address now in place
In foo	
Function prolog	
`push ebp`	save the previous frame register
`mov ebp, esp`	set register **ebp** to point to the frame base
`sub esp, 8`	reserve space in the stack for the variable **k**
Function body starts	
...	
`mov eax, dword ptr [ebp+8]`	example of accessing the argument **i**
`movsd qword ptr [ebp-8], xmm0`	example of accessing the variable **k**
...	
Function epilog starts	
`mov esp,ebp`	reset the stack pointer to a known value
`pop ebp`	restore the previous frame pointer
`ret`	return to the caller
Back in the caller	
`add esp,8`	restore the stack pointer

While the code might be changed in various ways when, for example, optimisation is applied or a different calling convention is used there is still a reasonable correlation between the resultant code and this model.

- Optimisers often re-use memory addresses for various different purposes and may make extensive use of registers to avoid having to read and write values to the stack.
- The **stdcall** calling convention used for the Win32 API itself slightly changes the function return: the called function is responsible for popping the arguments off the stack, but the basic principles are unchanged.
- The 'fastcall' convention passes one or two arguments in registers, rather than on the stack.
- 'frame pointer optimisation' re-purposes the **ebp** register as a general purpose register and uses the **esp** register, suitably biased by its current offset, as the frame pointer register.

In the 64-bit world while what happens from the programmer's view will be identical, the underlying implementation has some differences. Here is a summary of how the stack frame for **foo** might look when compiled as part of a 64-bit program:

Offset	Size	Contents
+72	1 byte	`char ch`
+64	4 bytes	`int i`
+56	8 bytes	return address
+32	8 bytes	`double k`

(Again, offsets can be taken from the assembler output). Notice that all the offsets are positive, and the smallest offset is 32. Perhaps even more surprising is that the stack frame size (from a debugger, the assembler output or by adding calls to a checking function) is **64** bytes; more than double the 24 bytes used in the 32-bit case. Why would this be the case – we might expect some of the items to double in size to match the word size but something else is going on here.

In the 64-bit calling convention the caller passes the first **four** arguments in registers but must reserve space on the stack for them. This provides a well-defined location for the called program to save and restore the corresponding argument if the value passed in the register would otherwise be overwritten.

Additionally space for four arguments is *always* reserved even when the function takes fewer than that. (These 32 bytes just above the return address on each function call are sometimes called the 'home space'.) So in our example, although we only have two arguments (**i** and **ch**) our caller will have reserved space for two other (unused) arguments. The full stack frame for **foo** can therefore be written as in the table overleaf.

The first offset in the stack frame is +32 because this function will in turn need to reserve stack space for up to four arguments when it calls another

by default both 32-bit and 64-bit applications are given a 1Mb stack

	Offset	Size	Contents
High mem	+96		*top of frame*
	+88	8 bytes	*reserved for 4th argument*
	+80	8 bytes	*reserved for 3rd argument*
	+73	7 bytes	*padding*
	+72	1 byte	`char ch`
	+68	4 bytes	*padding*
	+64	4 bytes	`int i`
	+56	8 bytes	*return address*
	+40	16 bytes	*padding*
	+32	8 bytes	`double k`
`rsp`*->	+0	32 bytes	*argument space for child functions*
Low mem			

* rsp normally remains here for the duration of the function.

function. So the function presets the stack pointer just below these four words to avoid having to modify the stack pointer when making function calls – it can just make the call.

The actual size of the first offset can be greater than 32 if, for example, more than four arguments are passed to a child function; but it can only be *less* if the function itself does not call any other functions.

Note that although we're only using 21 bytes of memory the stack frame is 64 bytes in size: that's over twice as much being wasted as being used. The 64-bit calling convention does, in general, seem to increase the stack consumption of the program. However, there are a couple of things that help to reduce the stack consumption.

- Firstly the 64-bit architecture has more registers (eight more general-purpose registers `r8` – `r15`). This allows more temporary results (or local variables) to be held in registers without requiring stack space to be reserved.
- Secondly, the uniform stack frame convention increases the number of places where a nested function call at the end of a function can be replaced with a jump. This technique, known as 'tail call elimination', allows the called function to 'take over' the current stack frame without requiring additional stack usage.

However, it still seems odd (at least to me) that Microsoft did not change the default stack size for applications when compiled as 64-bits: by default both 32-bit and 64-bit applications are given a 1Mb stack.

If your existing 32-bit program gets anywhere near this stack limit you may find the 64-bit equivalent needs a bigger stack (obviously this is very dependent on the exact call pattern of your program). This can be set, if necessary, by using the `/stack` linker option when the program is created – or even after the program has been linked using the same `/stack` option with the `editbin` program provided with Visual Studio.

This is a possible sequence of instructions for setting up the stack frame when `foo` is called in a 64-bit application:

In the caller	
`mov dl, 'c'`	set up low 8 bits of the `rdx` register with 'ch'
`mov ecx, 12`	set up low 32 bits of the `rcx` register with 'i'
`call foo`	enter `foo`, return address now in place
In foo	
Function prolog starts	
`mov byte ptr [rsp+16], dl`	save the second argument in the stack frame
`mov dword ptr [rsp+8], ecx`	save the first argument in the stack frame
`sub rsp,56`	reserve space for the local variables, and 32 bytes for when `foo` calls a further function
Function body starts	
...	
`mov eax, ecx`	example of accessing the argument `i`
`movsdx qword ptr [rsp+32], xmm0`	example of accessing the variable `k`
...	
Function epilog starts	
`add rsp, 56`	reset the stack pointer
`ret`	return to the caller
Back in the caller	
	nothing to see here ... move along

As you can see the 64-bit code is simpler than the 32-bit code because most things are done with the `mov` instruction rather than using `push` and `pop`. Since the stack pointer register `rsp` does not change once the prolog is completed it can be used as the pointer to the stack frame, which releases the `rbp` register to be a general-purpose register.

Note too that the 64-bit code only updates the relevant part of the register and memory location. This has the unfortunate effect that, if you are writing tools to analyse a running program or are debugging code to which you do not have the complete source, you cannot as easily tell the actual value of function arguments as the complete value in the 64-bit register or memory location may include artefacts from earlier. In the 32-bit case, when an 8bit char was pushed into the stack, the high 24bits of the 32-bit value were set to zero.

One other change in the 64-bit convention is that the stack pointer must (outside the function prolog and epilog) **always** be aligned to a multiple

'over-sized' variables will be located in working storage above the stack frame for the target function

of 16 bytes (not, as you might at first expect, 8 bytes to match the word size). This helps to optimise use of the various instructions that read multiple words of memory at once, without requiring each function to align the stack dynamically.

Finally note that the 64-bit convention means that the called function returns with the stack restored to its value on entry. This means function calls can be made with a variable number of arguments and the caller will ensure the stack is managed correctly.

Note that in Visual Studio 2013 Microsoft have added a second (explicit) calling convention for **both** 32-bit and 64-bit programs, the `__vectorcall` convention. This passes up to six 128bit or 256bit values using the SSE2 registers `xmm` and `ymm`. I'm not discussing this convention further – interested readers can investigate this by looking up the keyword on MSDN.

More on passing variables

The 64-bit bit convention dictates that the first four arguments are passed in fixed registers. These registers, for integral and pointer values are `rcx`, `rdx`, `r8` and `r9`. For floating point values the arguments are passed in `xmm0` – `xmm3`. (The older x87 FPU registers are not used to pass floating point values in 64-bit mode.)

If there is a mix of integral and floating point arguments the unused registers are normally simply skipped, for example passing a `long`, a `double` and an `int` would use `rcx`, `xmm1` and `r8`.

However, when the function prototype uses ellipses (i.e. it takes a variable number of arguments), any floating pointing values are placed in **both** the integral **and** the corresponding `xmm` register since the caller does not know the argument type expected by the called function.

For example, `printf("%lf\n", 1.0);` will pass the 64-bit value representing 1.0 in both the `xmm1` and `rdx` registers.

When a member function is called, the `this` pointer is passed as an implicit argument; it is always the first argument and hence is passed in `rcx`.

The overall register conventions in the x64 world are quite clearly defined. The documentation [Register Usage] describes how each register is used and lists which register values must be retained over a function call and which ones might be destroyed.

Bigger (or odder) values

Another change in the 64-bit calling convention is how larger variables (those too big for a single 64-bit register) or 'odd' sized variables (those that are not exactly the size of a `char`, `short`, `int` or full register) are passed. In the 32-bit world arguments passed by value were simply copied onto the stack, taking up as many complete 32-bit words of stack space as required. The resulting temporary variable (and any padding bytes) would be contiguous in memory with the other function arguments.

In the 64-bit world any argument that isn't 8, 16 32 or 64-bits in size is passed by **reference** – the caller is responsible for making a copy of the argument on the stack and passing the address of this temporary as the appropriate argument. (Note that this passing by reference is transparent to the source code.) Additionally the caller must ensure that any temporary so allocated is on a 16byte aligned address. This means that the temporary variables themselves will not necessarily be contiguous in memory – the 'over-sized' variables will be located in working storage above the stack frame for the target function. While this should very rarely affect any code it is something to be aware of when working with code that tries to play 'clever tricks' with its function arguments.

Local variables

The compiler will reserve stack space for local variables (whether named or temporary) unless they can be held in registers. However it will re-order the variables for various reasons – for example to pack two `int` values next to each other. Additionally arrays are normally placed together at one end to try and reduce the damage that can be done by a buffer overrun.

Return values

Integer (or pointer) values up to 64-bits in size are returned from a function using the `rax` register, and floating point values are returned in `xmm0`. Values other than these are constructed in memory at an address specified by the caller. The pointer to this memory is passed as a hidden first argument to the function (or a hidden second argument, following the `this` pointer, when calling a member function) and then returned in `rax`.

Debugging

In this example the first two instructions in the prolog save the argument values, passed in as register values, in the stack frame. When optimisation is turned on this step is typically omitted and the register values are simply used directly. This saves at least one memory access for each argument and so improves performance.

The compiler can now use the stack space for saving intermediate results (which reduces its need for other stack space) knowing there will always be space for four 64-bit values on the stack.

Unfortunately this performance benefit comes at the price of making debugging much harder: since the function arguments are now only held in (volatile) registers it can become hard to determine their values in a debugger (or when examining a dump). In many cases the value has simply been lost and you have to work back up the call stack to try and identify what the value *might* have been on entry to the function.

While this sort of optimisation is common in 32-bits for *local* variables the fact the arguments are (usually) passed on the stack does increase their longevity. For example, consider this simple function:

```
void test(int a, int b, int c, int d, int e)
{
  printf("sum: %i\n", a + b + c + d + e);
}
```

If I build a program with an optimised build of this function and breakpoint on the `printf` statement, in a 32-bit application I can still see the values

of the arguments (when using the default calling convention, unless the optimiser has made the entire function inlined). If I try the same thing with a 64-bit optimised build I get, for example:

Argument	Actual value passed	Value displayed in the debugger
a	11111	1
b	22222	1067332614
c	33333	1
d	44444	4058400
e	55555	55555

As you can see, the fifth argument is displayed correctly because, as described above, only the first four arguments (in Windows 64-bit) are passed in registers. In general the debugger cannot locate the values from the register (even assuming the value is still there!) and so it simply displays what is in the stack frame location reserved for that argument; but as the argument has not been persisted to the stack frame in the release build the contents are arbitrary.

In some cases, for example where the argument is a pointer value, the arbitrary value can even break the debugger (at least in VS 2012).

This can make it significantly harder to identify the reason for a failure in a build of an application where some or all of the code is compiled with optimisation. This is a particular problem when you get a dump of a production system, where reproducing the fault yourself on a non-optimised build may be hard.

Stack walking

The other main area where the 64-bit calling convention differs from the 32-bit one is when walking the stack. There are two main cases when the stack is walked.

- Handling an exception
- In various debugging scenarios

In the 32-bit world these two cases were handled very differently.

For exceptions, each thread in the Win32 subsystem contained a singly-linked list of exception handlers, maintained in the stack with the address of the first handler held in the thread environment block.

Additionally the MSVC compiler maintained a simple state machine for each function containing exception handling logic (either implicit or explicit) and used the state variable, which was also held in the stack frame, during unwinding of the stack when handling an exception.

This state variable was used by the exception handling logic, in conjunction with some other tables built into the binary image by the compiler, to find the catch handler, if any, for the thrown exception and also to identify the completely constructed objects on the stack that should be destroyed when unwinding the stack.

There were at least three main problems with this approach.

- The exception code was fragile under accidental or malicious stack overwrites
- Management of the exception chain had a measurable performance impact
- The stack frame was larger, again impacting performance (among other things)

For the various debugging scenarios the simplest approach was to follow the chained base pointers: the value of the `ebp` register provides the address of the current frame. Each frame was expected to have the entry at +0 containing the previous base pointer and the entry at +4 to contain the return address.

This mechanism was, like the exception chain, quite fragile and was also complicated by the 'frame pointer optimisation'. The MSVC toolchain would add additional data to the debugging files (with the pdb extension) for each module containing information to enable a debugger to locate the stack frame even when this optimisation was enabled, but this required the PDB files to be present and accessible to allow the stack to be reliably walked. The tables were also quite slow to access, meaning their use was unfeasible in some of the places where stack walking at runtime was used (such as tracking memory allocations and deallocations).

As you may know, although the frame pointer optimisation does produce a performance benefit (normally a low single digit percentage), Microsoft have disabled it in their operating system builds since Windows XP service pack 2 as they considered the increased ability to debug production problems was more significant than the loss of performance.

The 64-bit convention uses a pure table-based system that is linked into the binary image and is used to walk the stack in **both** the cases above. This has several benefits. Firstly, there is improved robustness and security since the data structures are in read-only memory rather than created on the stack. Secondly, there is a small performance improvement as the tables are fixed and only accessed if and when stack walking is required. Thirdly, since the data structures are held in the binary itself, stack walking is reliable even if the PDB file is not present.

The instruction pointer is used to find the currently executing image and the offset into that image. This offset is then used to look up the correct entry for the current function in the tables for the current module. The table entries contain, among other things, a description of the stack frame for each function: in particular which register contains the base address and what the vital offset values are. This information allows the stack walker to reliably walk up the list of stack frames to identify the calling functions and/or find the correct exception handler without relying on data tables held in the stack itself.

While not quite as fast a simply chaining up a linked list of exception records or frame pointers the mechanism is fast enough to be used at runtime. The Win32 API exposes a method, `CaptureStackBackTrace` that understands the data structures involved and can be used by application programs to capture the address of the functions in the call stack.

There are further functions providing support for this stack walking: such as `RtlLookupFunctionEntry` which obtains a pointer to the relevant data for a specific address; but I recommend that you use the supplied stack capture function: as while the data structures are (at least partly) documented making correct use of them is not for the faint hearted.

If you need to write 64-bit *assembler* code then you need to ensure that these tables are correctly built. There are a number of restrictions as to the instructions that can be used in the function prolog and epilog; and additional assembler directives must be written to ensure the assembler generates the correct data structures.

An even more complicated activity is when you need to generate executable code on the fly at runtime. The Win32 API provides a function `RtlAddFunctionTable` that you can use to dynamically add function table entries to the running module. Unfortunately there do not seem to be any helper functions to facilitate building up the required data structures.

Additionally, it can be hard to verify that the data structures are in fact correct – the first indication that they are incorrect may occur when the system is trying to handle an exception as, if it is unable to correctly process the function table entry for your dynamically created code this will almost certainly result in unexpected program termination.

However, since the same control structures are used for walking the call stack in the debugger as are used when an exception is thrown, some checking can be done by verifying the call stack displays correctly in a debugger as you step through the generated code.

Dump busting

While in general the data tables in the binary images do provide a very reliable way to walk the stack, the technique can fail when processing a dump if the actual binaries are not accessible to the debugger that is reading the dump file.

As an example, consider a very simple function that throws an exception:

```
void func()
{
  throw 27;
}
```

If we package this function in a DLL, and call this DLL from a main program, the exception handling logic walks up the chain from the site of the exception (which is actually the **`RaiseException`** function inside `kernelbase.dll`) to the handler, if any, in the calling function. At the time of the exception all these tables are present in the executing binaries; but if a minidump is taken then the code modules may well not be included in the dump. This will depend on which options are used when the minidump is created, but space is often at a premium and so a complete memory dump may not be realistic.

Let us look at what happens when the resulting dump is processed on a *different* machine where not all the binaries are present: in this example we have access to a copy of the `main.exe` program but not of the `function.dll`. In the 32-bit world the absence of the binaries simply means the function names are missing, but the stack walk itself can complete (I've disabled FPO in this example).

The 32-bit exception in a debugger

- **`KERNELBASE!RaiseException+0x58`**
- **`function.dll!_CxxThrowException(void * pExceptionObject, const _s__ThrowInfo * pThrowInfo) Line 152`**
- **`function.dll!func() Line 4`**
- **`main.exe!main() Line 12`**
- **`main.exe!__tmainCRTStartup() Line 241`**
- **`kernel32.dll!@BaseThreadInitThunk@12 ()`**
- **`ntdll.dll!___RtlUserThreadStart@8 ()`**
- **`ntdll.dll!__RtlUserThreadStart@8 ()`**

The same exception, examined from a minidump on another machine

- **`KERNELBASE!RaiseException+0x58`**
- **`function+0x108b`**
- **`function+0x1029`**
- **`main.exe!main() Line 12`**
- **`main.exe!__tmainCRTStartup() Line 241`**
- **`kernel32.dll!@BaseThreadInitThunk@12 ()`**
- **`ntdll.dll!___RtlUserThreadStart@8 ()`**
- **`ntdll.dll!__RtlUserThreadStart@8 ()`**

If the same operations are performed with a 64-bit build of the same program the results when reading the dump are quite different.

The 64-bit exception in a debugger

- **`KernelBase.dll!RaiseException ()`**
- **`function.dll!_CxxThrowException(void * pExceptionObject, const _s__ThrowInfo * pThrowInfo) Line 152`**
- **`function.dll!func() Line 4`**
- **`main.exe!main() Line 11`**
- **`main.exe!__tmainCRTStartup() Line 241`**
- **`kernel32.dll!BaseThreadInitThunk ()`**
- **`ntdll.dll!RtlUserThreadStart ()`**

(Note that debuggers sometimes display slightly different line numbers – the entry for **`main.exe!main()`** from which **`func`** was called is shown as Line 11 in the 64-bit debugger (correctly) but as the next line, Line 12, in the 32-bit debugger.)

The same exception, examined from a minidump on another machine

- **`KERNELBASE!RaiseException+0x39`**
- **`function+0x1110`**
- **`function+0x2bd90`**
- **`0x2af870`**
- **`0x1`**
- **`function+0xf0`**
- **`0x00000001`e06d7363`**

The stack walk is unable to get before the first address inside `function.dll` in the absence of the function data tables. (What the debugger seems to do when the data is missing is to simply try the next few possible entries on the stack but it is very rarely successful in finding the next stack frame.)

Note that providing the PDB files in this case does not help to walk the stack correctly, as it is the DLL and EXE files that contain the stack walking data. It therefore becomes important to ensure that you can easily obtain the **exact** versions of the binary files that the target machine is using if you wish to successfully process the smaller format minidump files.

Conclusion

The 64-bit Windows calling convention does seem to be an improvement on the 32-bit convention, although I personally wish that a little more care had been taken to ensure that problem analysis was easier. As I have mentioned above, while the mechanism does offer increased performance, it also decreases the likelihood of successful problem determination (especially with an optimised build.)

It is my hope that some understanding of how the 64-bit convention works will aid programmers as they migrate towards writing more 64-bit programs for the Windows platform. ■

Acknowledgements

Many thanks to Lee Benfield, Dan Azzopardi and the Overload reviewers for their suggestions and corrections which have helped to improve this article.

Useful references

Microsoft x64 Calling Convention http://msdn.microsoft.com/en-us/library/9b372w95.aspx

Microsoft Macro Assembler Directives http://msdn.microsoft.com/en-us/library/8t163bt0.aspx

[Register Usage] http://msdn.microsoft.com/en-us/library/9z1stfyw.aspx

Teenage Hex

There's a big push to get programming into schools. Teedy Deigh considers what would suit the target audience.

Is P = NP? Why would you ever play Twister when sober? How do you get adolescents interested in programming? This last question is a simpler and more concrete instance of the larger problem of how to get adolescents interested in anything. It is hoped that any solution to this programming challenge (P) will allow parents, teachers and society to then solve the non-programming challenge (NP).

There has been much discussion of how to get teenagers interested in tech beyond their phones. The Raspberry Pi has helped support this trend by introducing a device named after fruit, following a venerable tradition of tech targeted at children, such as Tangerine, BlackBerry and Apple.

But if the goal is programming, what should be the programming language? In the 1980s BASIC was considered the language of choice and made a strong impression on a whole generation. The home computing boom succeeded where previous initiatives, such as Teenage CICS, were felt to have corporate undertones. Edsger Dijkstra observed, however, that

> It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

Dijkstra's insight does much to explain most of the code written in industry from the 1990s onwards.

These days Python is typically considered the language of choice, representing, as it does, a language that will set false expectations about what other mainstream programming languages are like in terms of feature set (orthogonal and considered), syntax (in space no one can hear you scream) and culture (humorous and surreal with a 1970s twist, washed through with British cynicism and Dutch De Stijl sensibilities). This dovetails nicely with the practice of teaching Haskell to computer science undergraduates in preparation for industry.

A more careful analysis of the target audience, however, suggests that Python may not be the most suitable choice. For example, the torpor of a typical adolescent suggests a preference for static rather than dynamic typing and lazy rather than strict evaluation.

If not Python, then what? As it's a software problem we already have a process (indeed, many processes) for helping to determine a solution: requirements gathering and specification. As it's a software problem, we already know that any discussion of requirements can be circumvented and left to people who failed to make the grade as programmers, washed up in dead-end jobs and made-up disciplines such as business analysis, where they are left to write stories and play cards. They can be humoured, encouraged and given the belief that their work has meaning and relevance, even when it is ultimately ignored. If there is one thing that may help to motivate adolescents into programming, it is being shown this kind of McJob. On the other hand, they may find much in common with their existing situation.

As it's a software problem we already know the universal solution: invent another programming language. Riding on the coat-tails of a previous fad, it can be considered a domain-specific language. In this case the domain is teenagers and the co-domain is programmers.

Beyond laziness and a general lack of dynamic, what other characteristics should such a language have? Laziness suggests something functional, although it is worth noting that teens object to most things. The typical object model, however, involves classes, which is often at odds with the politics of the target demographic. Following any kind of procedure is beyond reason, so classic imperative and procedural programming is unlikely to be a wise choice. And logic programming is clearly a non-starter.

The programming style chosen should be driven primarily by arguments, which lends further support to functional programming. The language need not, however, be a pure functional language. Given that most teenagers are mixed up, it seems appropriate to reflect this confusion in the language design. It will also enable them to employ words like multi-paradigm and postmodern with greater confidence in their media studies essays.

The language should capitalise on already familiar operators and concepts, such as the `Maybe` and `Whatever` monads and the `like` and `isKindOf` comparisons. Other relational operators would include `owns` instead of greater than, although, because fuzzy rather than Boolean logic should be used, this is not like an exact drop-in replacement, you know, right. The use of `null` is discouraged in modern language design; `Duh` is proposed as an alternative. Teen sensitivity to anything and everything can be acknowledged by ensuring the language is case sensitive, although a compelling case can also be made for case-indifferent syntax.

What of the language's execution model? A benefit of functional programming is the lack of side effects, so adolescents would be able to enjoy doing what they wanted without having to worry about the consequences. In contrast to many functional languages, however, a heavy reliance on exceptions is a likely need, although the exception model should be as layered as possible. Throwing up meets a common need.

Although the possibility for concurrency is intrinsic to many functional languages, it may be prudent not to include support for concurrency in a teaching language. The scheduling model favoured by most adolescents is at best frustrating, being typically pre-emptive but with long delays and without resumption of existing tasks, i.e., easily distracted. Any alternative scheduling algorithm is likely to be considered (so) unfair. Priority inversion would undoubtedly be a common problem, with things adults consider trivial taking precedence over things considered to be more important. Similarly, deadlock would hamper much progress.

In terms of development environment, an IDE would be unsuitable as there is little that is integrated about teenagers. Something that is console-based, preferably black-and-white, will most likely satisfy needs and neediness given the amount of time teenagers already spend consoling one another.

It almost goes without saying that test-driven development is completely anathema to the adolescent mindset, involving, as it does, a cautious and considered test-first approach, where desires are clearly specified in advance of their realisation. That does not, however, preclude unit testing in general. The approach most suited to teenagers is the test-after approach of plain ol' unit testing, i.e., POUTing.

And, finally, what of the design philosophy that should be taught? This choice, at least, is simple: KISS. ■

Teedy Deigh learned to program when the only visual thing about BASIC was the line numbering. She does not believe that BASIC has had any lingering effects on her abstraction-lite, GOTO-inflected coding style, although she confesses to occasionally missing the line numbers.