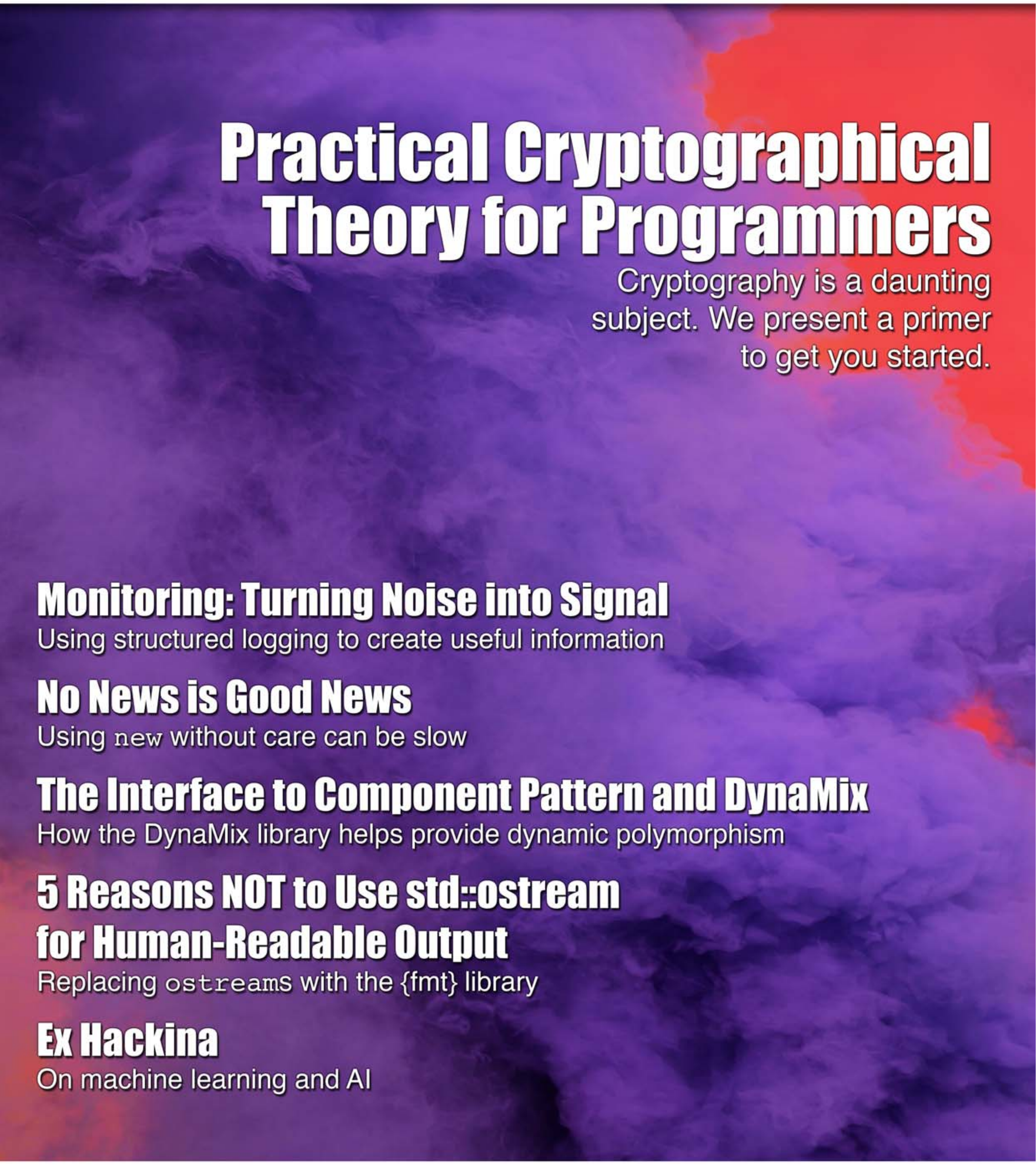
overload 144

APRIL 2018 £3



Practical Cryptographical Theory for Programmers

Cryptography is a daunting subject. We present a primer to get you started.

Monitoring: Turning Noise into Signal

Using structured logging to create useful information

No News is Good News

Using `new` without care can be slow

The Interface to Component Pattern and DynaMix

How the DynaMix library helps provide dynamic polymorphism

5 Reasons NOT to Use std::ostream for Human-Readable Output

Replacing `ostreams` with the {fmt} library

Ex Hackina

On machine learning and AI

A magazine of ACCU

ISSN: 1354-3172

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe

OVERLOAD 144

April 2018

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Balog Pal
pasa@lib.h

Ben Curry
b.d.curry@gmail.com

Paul Johnson
paulf.johnson@gmail.com

Klitos Kyriacou
klitos.kyriacou@gmail.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Philipp Schwaha
<philipp@schwaha.net>

Anthony Williams
anthony@justsoftwaresolutions.co.uk

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 145 should be submitted by 1st May 2018 and those for Overload 146 by 1st July 2018.

Overload is a publication of the ACCU

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

4 No News is Good News
Paul Floyd uses Godbolt's compiler explorer to see what happens when you use 'new'.

8 Monitoring: Turning Noise into Signal
Chris Oldwood shows the benefits of structured logging.

12 The Interface to Component Pattern and DynaMix
Borislav Stanimirov demonstrates the DynaMix library for dynamic polymorphsim in C++.

16 5 Reasons NOT to Use std::ostream for Human-Readable Output
Sergey Ignatchenko shows why the {fmt} library is better than std::ostream.

21 Practical Cryptographical Theory for Programmers
Deák Ferenc helps you get started with cyrpotgraphy.

28 Ex Hackina
Teedy Deigh takes the Turing test.

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.

Deeds not words

Women's suffrage used the motto "Deeds not Words". Frances Buontempo applies this to programming.

It's about time I got on and wrote an editorial. I have been making excuses for well over five years now! Well, I say that, and I am sure our readers all agree. It's much easier to say something than actually get on with it. I have previously considered why it can be important to talk things through as a team before rushing in to solve a problem or write code [Buontempo15]. Charles Tolman's recent mini-series also reminded us that changing code to speed it up, without careful measuring and thinking first, can make problems worse [Tolman17]. However, "There is a time for everything, and a season for every activity under the heavens... A time to search and a time to give up. A time to keep and a time to throw away." [Ecclesiastes 3:1]. There is a time for meetings and a time for coding. There is a time for words and a time for deeds.

This year sees the centenary of women being given the vote in the UK, a select few women to be fair. The internet tells me "Parliament passed an act granting the vote to women over the age of 30 who were householders, the wives of householders, occupiers of property with an annual rent of £5, and graduates of British universities." [Wikipedia-a]. The internet also tells me women could originally vote, or at least were not explicitly excluded, but the Reform Act stopped this. The act also stopped most men from voting. Landownership and occupation, and gender, became criteria for enfranchisement. During the First World War, many women took over roles usually performed by men, bringing the question of voting to the fore. War often brings about unexpected changes, from technological innovations, such as computers, to social reform! "There is a time for war and a time for peace." [Ecclesiastes 3:8]. I'm not suggesting war is a good thing, rather that the effects are interesting and unexpected. Who would have predicted a war would start conversions about women's suffrage or voting? Calling members of the Women's Social and Political Union 'suffragettes' seems to have been a deliberate insult [Oxford Dictionaries], adding the diminutive ette to suffrage to subtract from the movement. The word now sticks and the original connotation is lost. Words do matter, but sometimes deeds are more important or more likely to get results. I believe "Deeds not words" became one of the suffragettes' slogans while campaigning via hunger strikes, riots and similar [Parliament.uk]. Do you have a motto or slogan? After a long discussion about a comment that did not match some code in a role a long while ago, I used "Delete the comment and fix the code" as a motto for a while. I still maintain this is a worthwhile life goal. It's not as extreme as starting riots, but a form of direct action is sometimes required. There are times when words are not enough.

Words come in many forms, sometimes written, sometimes in a telco, sometimes face to face, or even via webcasts. Let's talk about meetings. How many of you get stuck in meetings, of increasing length and frequency, nearer a release in order to talk about the impending doom? Or success? Do you have lots of meetings before you even begin a software project? It can be frustrating to sit through a meeting talking about software without having a chance to experiment and explore first, so that decisions and action points can be based on knowledge rather than speculation. Speculating is important, but does not require a meeting, rather a gathering to devise, plot and dream. Of course, you need to meet up and share what you have found out. You do need meetings, but not all the time! I am stuck in a form of personal meeting madness. I even visited Chichester cathedral on holiday the other week and was shocked to see a large gathering in the nave. It turned out to be a Women's Institute and flower arrangers' meeting, plotting a flower festival later in the year. I have no idea how or why I end up in so many meetings. Or why so many of them go on for so long. Wandering into the flower arranger's meeting was an absurdness that amused me.

Assuming you manage to sanitise your meetings, keeping them on time and on topic, keeping irrelevant chatter for the water cooler or the pub, what other ways do words get in the way of deeds? A perennial problem of documentation persists for many programmers. This is related to the tension over meetings that has been tormenting me recently. Managers and product owners want to be kept up to date with the current state of play, and understand what choices have been made, so they can steer development and keep things on track. Developers also want to write working software, but if they spend all day in meetings they can't do their job. If the developers refuse to attend meetings, the managers can't do their job. You can find a balance point to keep everyone happy and the project moving along, but it takes work. Similarly, managers tend to request documentation. They don't want to have to wade through thousands of lines of code to understand what a product does or how it works. A developer, on the other hand, might just break down in tears if confronted with fifty or so pages of documentation 'summarising' how 50 lines of code works, particularly if they can't find the code because the latest version isn't in version control. What gives?

Automatic generation of documentation seems like the best way to solve this problem. Most of us have encountered Doxygen at some point [Doxygen]. It can be really useful. However, there is no substitute for a short readme file, showing how to build and run the code. Automatically generated documentation is very unlikely to include this. I have been working with Chris Simons, preparing a workshop for this year's ACCU conference [ACCU18]. We will demonstrate how to use a Java framework for evolutionary computing, which Chris has written about previously [Ramirez17]. The framework is very powerful; however, the main documentation consists of a 65 page pdf, replete with UML

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

diagrams and worked examples and there is a doxygen generated webpage. To be fair, there are notes on how to build and run the examples! Despite all this, I am none the wiser as to what BLXAlphaCrossover is [Sourceforge]. Obviously, it is a crossover operator, that's what the X means (you guessed that, right?), using an alpha (a parameter I presume). A relatively thorough internet search suggests BL stands for 'blended'. I think it amounts to subtraction and multiplying and I promise I will know by the conference. I shall look at the source code. There is documentation. It has diagrams. It shows me how to use the framework. It includes BLXAlphaCrossover but doesn't tell me what it does, rather what it derives from and its methods. The details themselves are in the code. No matter how thorough you have been, sometimes only the code itself tells the whole story. *97 Things Every Programmer Should Know* [Henney10] points this out. Documentation may tell us what code is supposed to do, but the code tells us what it actually does. A design document may tell us the intentions, but not what actually happened when to code was written. Sometimes you need to try out the code, even trying a few scratch refactors [Feathers04], to understand how it works. Extract methods, move variables, change access levels to see what happens. Sitting in a presentation watching someone point at a UML diagram is no substitute. Reading a pdf or wiki is no substitute. You need to get your hands dirty to fully understand.

Once you have your hands on the code you can find out all kinds of things. When the documentation doesn't show you exactly what a function is up to, there is sometimes a comment, which fails to make it to the doxygen output, referring to an academic paper. Many people regard comments as documentation. They can be, though they can get out of sync with the code. A readme showing me how to run the examples would not explain the minutiae of details concerning how to approach combining real values from two parent solutions in a genetic algorithm (the blending alluded to previously).Having to dig inside source code to find what it all means renders a long pdf or word document somewhat superfluous. Another trick is searching the internet for the comments you find in a library. On many occasions, the comment turns up with other source code, along with better documentation or even unit tests. Never fix a comment – even if it contains a spelling mistake. You are destroying forensic evidence. You will regret this. I know I suggested, "Delete the comment and fix the code" earlier. There is a time and a place.

Documentation after the fact is important. Different people, with differing roles, have different requirements. It is important to keep everyone happy. You may disagree, and might think that's why I keep getting stuck in long meetings. We'll save that for a pub discussion. Don't be shy about automatically generating your documentation if someone insists they need it. Do provide me with a small file showing how to get up and running quickly though! Please! Now, step back in time. What do you expect to see in writing before you start on a greenfield project? A two-hundred-page upfront specification, along with a Gantt chart? User stories, along with the stick figures? Some BDD-style user requirements? Millions of meeting invites to flesh out the pipe dream? A vague mission statement and contact details for product owners and the rest of the team? A list of allowed libraries or programming languages? An invitation to a training session on how to use your workstation safely, including how to sit on a chair? What do you need to know in order to get things done? What is the one thing you cannot do without, your sin qua non for success? [Wikipedia-b]. I need to know how we will verify or test a solution is working. You may have different priorities. That's ok.

If you are part of a team, you might find other people have other needs and assumptions. I alluded to the tension between people with different roles earlier. A manager in need of useful, high level documentation and a dev in need of a sane development pipeline might be trying to build the same product, but might have different assessments of how to achieve this in an efficient manner. When people with different roles don't understand each other, the tension can lead to conflict and all-out war. Or lots of long meetings. Alexei Sayle, in 'Thatcher stole my trousers' [Sayle16], claims "No good ever came from meetings. The Russian revolution was just a meeting that got out of hand." One viewpoint. How can you learn to communicate as a team and build up consensus? Sometimes pairing on a task can help. If you can demonstrate why it takes several days to build code to replicate what a spreadsheet built in an hour does, you might get some understanding. If you go with your manager to the senior managers' meeting, just once, you might realise why there is an insistence on a Gantt chart or spreadsheet. If you sit with your testers or support team, you might realise why they want documentation including a list of error messages. If you sit with one of your colleagues and pair program a prototype with them, you will also learn lots. Building a prototype gives you something concrete to discuss in the next meeting. Deeds, not words, can help you make progress. If you intended to throw away the prototype and find it ends up being the basis for a large production system, you have a different problem.

If you are going round and round in circles, never writing an editorial, an article for *Overload* or even code, remember Poincaré's recurrence theorem: "certain systems will, after a sufficiently long but finite time, return to a state very close to the initial state" [Wikipedia-c]. This applies to mechanical systems under some specific conditions, but can be misappropriated to apply to software development. If you start with lots of meetings, you will end with lots of meetings. If you need an algorithm to do something specific, someone has probably already written it. If you rewrite a system, because the original code was horrific, your re-write will eventually be re-written. If you attend training on how to sit on a chair, you will probably end up having further similar training.

What has been will be again,
what has been done will be done again;
there is nothing new under the sun.
~ Ecclesiastes 1:9

References

[ACCU18] ACCU Conference 2018: https://conference.accu.org/2018/sessions.html#XSimplytheBestOptimisingwithanEvolutionaryComputingFramework

[Buontempo15] Buontempo, Frances (2015) 'A little more conversation, a little less action' *Overload*, 23(127):2–3, June 2015. https://accu.org/index.php/journals/2106

[Doxygen] Doxygen www.doxygen.org

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

[Henney10] Henney, Kevlin (ed.) (2010) *97 Things Every Programmer Should Know* http://programmer.97things.oreilly.com/wiki/index.php/Only_the_Code_Tells_the_Truth O'Reilly

[Oxford Dictionaries] 'Woman or suffragette' https://blog.oxforddictionaries.com/2013/05/02/woman-or-suffragette/

[Parliament.uk] 'Deeds not words' http://www.parliament.uk/about/living-heritage/transformingsociety/electionsvoting/womenvote/overview/deedsnotwords/

[Ramirez17] 'Evolutionary Computing Frameworks for Optimisation', *Overload* #142 – December 2017, Aurora Ramírez and Chris Simons https://accu.org/index.php/journals/2444

[Sayle16] Sayle, Alexi (2016) 'Thatcher stole my trousers', 2016 Bloomsbury Circus

[Sourceforge] BLXAlphaCrossover: http://jclec.sourceforge.net/data/jclec4-classification-doc/net/sf/jclec/realarray/rec/BLXAlphaCrossover.html

[Tolman17] Tolman, Charles (2017) 'A Design Example' *Overload* #142 – December 2017 https://accu.org/index.php/journals/2447

[Wikipedia-a] 'Women's suffrage in the United Kingdom' https://en.wikipedia.org/wiki/Women%27s_suffrage_in_the_United_Kingdom

[Wikipedia-b] Sine qua non: https://en.wikipedia.org/wiki/Sine_qua_non

[Wikipedia-c] Poincaré recurrence theorem: https://en.wikipedia.org/wiki/Poincar%C3%A9_recurrence_theorem

No News is Good News

Using 'new' without care can be slow. Paul Floyd uses Godbolt's compiler explorer to see what happens when you do.

There are two influences that have inspired me to write this article. The first is that I've been playing a lot with Compiler Explorer (https://godbolt.org). Secondly, a while back I read *Optimized C++* by Kurt Guntheroth. It contains a chapter on using dynamic memory (Chapter 6: Optimize Dynamically Allocated Variables).

I agree with a lot of what is said. In short, there is a description of the types of memory available in C++ (automatic, dynamic and static); a description of how this memory relates to variables in code; APIs to deal with dynamic memory; smart pointers and many tips on optimizations related to dynamic memory. In this article I'm going to explore why you should be trying to optimize use of **new** by digging down to the machine code.

When I'm looking at production C++ code I do see a lot of gratuitous uses of **new**, for instance

```
list <handle>* handleList = new list <handle>;
...
processList(handleList);
```

I suspect that there are a few possible reasons for writing such code:

- the influence of Java
- the assumption that where an API takes a pointer, you have to pass it a pointer allocated on the heap
- not realizing that, at least for standard library containers, the object itself is quite small and that the bulk of the memory (what is contained) is dynamically allocated. A **std::list** is only 24 bytes, for instance (on 64bit Linux with GCC).

Obviously, in the second case the memory doesn't need to be allocated dynamically. The top-level object can perfectly well be on the stack. For instance, the above example could have been written:

```
list <handle> handleList;
...
processList(&handleList);
```

In addition, if I could change the interface to the **processList** API, I would almost certainly change it to take a reference rather than a pointer. Furthermore, **std::list** is rarely a good choice when it comes to performance, so I'd probably also change that if I could.

Both versions of the code do the same thing. So, what is wrong with the first version?

Memory refresher

For those of you who are a bit rusty on what the difference is between dynamic and automatic memory (I'll skip static), here is a quick refresher (Wikipedia has a longer description with diagrams [Wikipedia]). Firstly, they are also commonly also known by other names, referring to how they are often implemented. Dynamic allocation is also known as heap allocation, and automatic allocation is known as stack allocation. The stack is a large block of memory. It is referred to by CPU registers such as the Stack Pointer. Memory can be 'allocated' on the stack very quickly simply by manipulating the stack registers. The main drawbacks of stack memory are that it does not persist beyond the current scope and it can be quite limited in size. Heap memory is a separate block of memory, but this time it is controlled via functions like **malloc** [C++Ref-a] and **operator new** [C++Ref-b]. It's not so limited in size and persists until explicitly deallocated.

Paul Floyd Paul has been writing software, mostly in C++ and C, for about 30 years. He lives near Grenoble, on the edge of the French Alps and works for Synopsys developing tools for estimating the power consumption of electronic circuits. He can be contacted at pjfloyd@wanadoo.fr

Problems with new

Performance

There are a couple of reasons why heap allocation has worse performance than stack allocation.

- More memory is required – the allocator generally needs a small amount of memory for housekeeping in addition to the memory requested by the caller. In addition, the amount of memory might get rounded up to a larger size like the nearest power of 2 whilst stack allocation probably only rounds up to the nearest machine word size. On 64bit Linux, allocations have a 16byte minimum size and there is an 8byte housekeeping overhead.
- The biggest difference is that functions like **new** and **delete** are relatively slow. Much effort by library and OS writers has been made to make them as fast as possible (for instance, see the history of jemalloc [github-a]).

There is a third question, concerning the size of code that gets generated. This complicates the picture because it isn't always an apples and pears comparison. When you use stack allocation, it generally means that you are using RAII and the compiler will generate the code necessary for clean-up at the end of the scope. When you use heap allocation with raw pointers as above then it's up to you to ensure that resources get freed.

About the example code

In the examples that follow, I will continue to use **handleList**. In my testing I defined **Handle** to be

```
class Handle {
public:
  int data;
};
```

It doesn't matter what **Handle** is. The only thing of importance is to consider that **handleList** itself is something that needs some memory. I'm going to stick with the **std::list** in the examples for two reasons. Firstly, I want there to be a code smell. Secondly and more seriously, though the examples presented here are trivial I don't want them to be so small that the compiler optimizes them to almost nothing. Also, for that reason I've added calls to an externally defined **populateList** function.

	processStack():	
1	push r12	A: Function entry prologue
2	push rbp	
3	push rbx	
4	sub rsp, 48	
5	lea rbp, [rsp+16]	
6	mov QWORD PTR [rsp+32], 0	B: Inlined std::list construction of handleList on stack
7	mov QWORD PTR [rsp+8], rbp	
8	mov rdi, rbp	
9	movq xmm0, QWORD PTR [rsp+8]	
10	punpcklqdq xmm0, xmm0	
11	movaps XMMWORD PTR [rsp+16], xmm0	
12	call populateList()	C: Call function
13	mov rdi, QWORD PTR [rsp+16]	D: Check result and inlined destructor
14	cmp rdi, rbp	
15	je .L1	
16	.L3: mov rbx, QWORD PTR [rdi]	
17	call operator delete(void*)	
18	cmp rbx, rbp	
19	mov rdi, rbx	
20	jne .L3	E: Function exit epilogue
21	.L1: add rsp, 48	
22	pop rbx	
23	pop rbp	
24	pop r12	
25	ret	
26	mov rdi, QWORD PTR [rsp+16]	F: Stack Unwind Handling
27	mov rbx, rax	
28	.L6: cmp rdi, rbp	
29	je .L5	
30	mov r12, QWORD PTR [rdi]	
31	call operator delete(void*)	
32	mov rdi, r12	
33	jmp .L6	
34	.L5: mov rdi, rbx	
35	call _Unwind_Resume	

Table 1

The assembly was generated on Compiler Explorer using GCC 7.3 64bit.

Comparison of allocation methods

Digging deeper into the different allocation methods, if we have a stack allocation function like this:

```
void processStack()
{
  list<Handle> handleList;
  populateList(&handleList);
}
```

the optimized machine code that gets generated is in Table 1

Note that for non-exceptional flow, only items in blocks A to F in column 3 are executed. Block F only gets called when exceptions are thrown via the stack unwinding mechanism.

	processHeap():	
1	push rbp	A: Function Entry Prologue
2	push rbx	
3	mov edi, 24	
4	sub rsp, 8	
5	call operator new(unsigned long)	B: Dynamic allocation and inlined std::list constructor
6	mov rbx, rax	
7	mov rdi, rax	
8	mov QWORD PTR [rax+16], 0	
9	mov QWORD PTR [rax], rax	
10	mov QWORD PTR [rax+8], rax	
11	call populateList()	C: Call function
12	mov rdi, QWORD PTR [rbx]	D: Inlined std::list destructor, function exit epilogue and tail optimized delete
13	cmp rbx, rdi	
14	je .L12	
15	.L13: mov rbp, QWORD PTR [rdi]	
16	call operator delete(void*)	
17	cmp rbx, rbp	
18	mov rdi, rbp	
19	jne .L13	
20	.L12: add rsp, 8	
21	mov rdi, rbx	
22	mov esi, 24	
23	pop rbx	
24	pop rbp	
25	jmp operator delete(void*, unsigned long)	

Table 2

On the other hand, for heap allocation that does not ensure proper clean-up, like this:

```
void processHeap()
{
  list<Handle>* handleList = new list<Handle>;
  populateList(handleList);
  delete handleList;
}
```

the machine code that gets generated is in Table 2.

Thus, it has lost exception safety but made the generated code slightly shorter.

To regain exception safety with heap allocation, still using raw pointers, we would need to write something like Listing 1.

That doesn't look too pretty. Please don't do this at home. The generated machine code for this is in Table 3.

Moving on quickly, let's consider a fourth alternative, using a smart pointer.

```
void processHeapSmartPtr()
{
  auto handleList = make_unique<list<Handle>>();
  populateList(handleList.get());
}
```

OK, that's code that I could live with.

Note that if you want to use only the smart pointer and not the underlying raw pointer you would have to rewrite or overload `populateList`. When I tried this, I noticed that passing a reference to the `unique_ptr` prevented the compiler from inlining and optimizing the pointer use resulting in more register use and larger code. Furthermore, the

CppCoreGuidelines discourage passing references to smart pointers in cases like this [github-b]. When **`get()`** is used, there is no interface issue and the code size is barely any larger than the stack version.

The code flow in this case is quite similar to **`processStack`**.

The machine code for this function is in Table 4.

You may have noticed that the non-exception path for the three versions using the heap are the same (lines 1 to 25 and blocks A to D in the tables of assembler). The only thing that is different is how they handle exceptions.

Here is a summary of the size of the code generated. The byte sizes were obtained by **`nm`**

Performance

I did some measurements of these 4 functions. I just wrote a **`main()`** with a loop running a million times calling the 4 functions with empty stub **`populateList`** functions.

With Valgrind callgrind, I got the following numbers of op-codes executed per call.

Function	Op-codes executed
processStack	23
processHeap	27
processHeapNoLeak	27
processHeapSmartPointer	29

These are the exclusive counts i.e., only for the functions themselves. Whilst callgrind counts every instruction, by default it doesn't record functions that take less than 1% of the total. So, adding a loop is an easy way to ensure that they are included in the output. I get a slightly higher count for **`processHeapSmartPointer`** as I was doing these tests with GCC trunk, I expect that if I'd used GCC 7.3 the count would have been the same as the other two Heap functions.

This is pretty much what I was expecting

1. **`processStack`** is the fastest but not the smallest due to the exception handling
2. **`processHeap`** is the smallest because it does no exception handling
3. All of the functions using the heap execute similar numbers of machine instructions.

The picture is very different for the inclusive counts, that is the functions plus any callees.

Function	Op-codes executed
processStack	24
processHeap	360
processHeapNoLeak	360
processHeapSmartPointer	362

```
void processHeapNoleak()
{
    list<Handle>* handleList = nullptr;
    try
    {
        handleList = new list<Handle>;
        populateList(handleList);
        delete handleList;
    }
    catch (...)
    {
        delete handleList;
        throw;
    }
}
```

Listing 1

	processHeapNoleak():	
1	push rbp	A: Function Entry Prologue
2	push rbx	
3	mov edi, 24	
4	sub rsp, 8	
5	call operator new(unsigned long)	B: Dynamic allocation and inlined std::list constructor
6	mov rbx, rax	
7	mov QWORD PTR [rax+16], 0	
8	mov rdi, rax	
9	mov QWORD PTR [rbx], rax	
10	mov QWORD PTR [rbx+8], rax	
11	call populateList()	C: Call function
12	mov rdi, QWORD PTR [rbx]	D: Inlined std::list destructor and tail optimized delete
13	cmp rbx, rdi	
14	je .L20	
15	.L21: mov rbp, QWORD PTR [rdi]	
16	call operator delete(void*)	
17	cmp rbx, rbp	
18	mov rdi, rbp	
19	jne .L21	
20	.L20: add rsp, 8	
21	mov rdi, rbx	
22	mov esi, 24	
23	pop rbx	
24	pop rbp	
25	jmp operator delete(void*, unsigned long)	
26	mov rdi, rax	E: Exception handling, inlined std::list destructor delete and rethrow
27	call __cxa_begin_catch	
28	.L23: call __cxa_rethrow	
29	mov rdi, rax	
30	call __cxa_begin_catch	
31	mov rdi, QWORD PTR [rbx]	
32	.L19: cmp rbx, rdi	
33	je .L24	
34	mov rbp, QWORD PTR [rdi]	
35	call operator delete(void*)	
36	mov rdi, rbp	
37	jmp .L19	
38	mov rbx, rax	
39	call __cxa_end_catch	
40	mov rdi, rbx	
41	call _Unwind_Resume	
42	.L24: mov esi, 24	
43	mov rdi, rbx	
44	call operator delete(void*, unsigned long)	
45	jmp .L23	

Table 3

There are two things that stand out

- The number of op-codes executed hasn't change for the stack version, other than the call to the empty stub.
- The 3 heap versions have essentially the same count.

	processHeapSmartPtr():	
1	push r12	A: Function Entry Prologue
2	push rbp	
3	mov edi, 24	
4	push rbx	
5	call operator new(unsigned long)	B: Dynamic allocation and inlined std::list constructor
6	mov rbx, rax	
7	mov QWORD PTR [rax+16], 0	
8	mov rdi, rax	
9	mov QWORD PTR [rbx], rax	
10	mov QWORD PTR [rbx+8], rax	
11	call populateList()	C: Call function
12	mov rdi, QWORD PTR [rbx]	D: Inlined std::list destructor and tail optimized delete
13	cmp rbx, rdi	
14	je .L33	
15	.L34: mov rbp, QWORD PTR [rdi]	
16	call operator delete(void*)	
17	cmp rbx, rbp	
18	mov rdi, rbp	
19	jne .L34	
20	.L33: mov rdi, rbx	
21	mov esi, 24	
22	pop rbx	
23	pop rbp	
24	pop r12	
25	jmp operator delete(void*, unsigned long)	
26	mov rdi, QWORD PTR [rbx]	E: Stack unwind handling, inlined std::list destructor
27	mov rbp, rax	
28	.L37: cmp rbx, rdi	
29	je .L36	
30	mov r12, QWORD PTR [rdi]	
31	call operator delete(void*)	
32	mov rdi, r12	
33	jmp .L37	
34	.L36: mov rdi, rbx	
35	mov esi, 24	
36	call operator delete(void*, unsigned long)	
37	mov rdi, rbp	
38	call _Unwind_Resume	

Table 4

As usual, you may get different results on different platforms/compilers – I tried a couple of others and the results were broadly similar. Furthermore, this test case just uses stub functions. With real functions that actually do something the cost of the heap allocation would become relatively smaller. That said, the stack allocation here is around 15 times faster.

Conclusions

I think that the case is more or less settled. Use stack allocation where you can. It's safer, faster and requires writing the least code. Obviously, there are times when you need heap allocation:

- when you have huge data
- when you need extended object lifetime
- when you have self-referential data structures like graphs
- when you want to decouple interfaces like with the pimpl idiom
- when you have deeply recursive functions.

When you have to use heap allocation, use smart pointers. There is a small code size overhead, but if you use `make_unique` (or `make_shared`) then the difference in time performance is negligible compared to using smart pointers with the benefit of not having to worry about resource leaks. ■

Acknowledgements

Thanks to the reviewers for pointing out my omissions and inconsistencies.

Thanks also to Matt Godbolt for providing Compiler Explorer.

References

[C++Ref-a] malloc: http://en.cppreference.com/w/cpp/memory/c

[C++Ref-b] operator new: http://en.cppreference.com/w/cpp/memory/new

[github-a] jemalloc: https://github.com/jemalloc/jemalloc/wiki/Background

[github-b] CppCoreGuidelines: http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#r30-take-smart-pointers-as-parameters-only-to-explicitly-express-lifetime-semantics

[Wikipedia] Data segment: https://en.wikipedia.org/wiki/Data_segment

Monitoring: Turning Noise into Signal

Creating useful logging is a constant challenge. Chris Oldwood shows us how structured logging helps.

The humble text-based log file has been with us for a very long time and it often still suffices for simple tasks which will require little genuine support. But over the years systems have gotten so much bigger as the need to scale out has compensated for the inability to continually scale up. Consequently, the volume of diagnostic data has grown too as we struggle to make sense of what's going on inside our 'black boxes'.

Moving a system from the relative comfort of an internal data centre out onto the cloud also brings with it a host of new experiences which will add to the general ambient noise as networks routes go up and down and servers come and go on a regular basis, and I'm not talking about continuous deployment here either. Okay, it's not quite that unstable at the smaller end of the spectrum but there are many more reasons why your service does not appear to be behaving itself over-and-above the mistakes that we might make ourselves.

With that growing volume of diagnostic data we need to remain smart about how we handle it if we are to understand it. What might initially seem like just a bunch of random failures may in fact be something much easier to explain if only we chose to drill into it and categorise it further. We refactor our functional code to help us make more sense of the design and the same is true for our diagnostic code too – we need to continually refine it as we learn more about the problems we encounter so that they are easier to monitor and triage in future. Ultimately you do not want to be woken in the middle of the night because you cannot distinguish between a failure on your side of the fence and a known limitation of an upstream system which you cannot control.

Hence this article is about the journey you might take after starting with a big ball of monitoring data so that you can start to make sense of it, both inside and outside your team, by improving the code so that what was once potentially just noise can be turned into more useful signals.

Structured logging

The most basic form of log file is just a simple collection of lines of freeform text. These generally form a narrative which tries to tell a story about what's going on inside, e.g.

```
2018-02-16 08:14:01.100 9876 INF Fetching orders
2018-02-16 08:14:01.123 9876 ERR Failed to fetch
orders for '1234'
2018-02-16 08:14:01.145 4225 PRF Orders request
took 10 ms
```

With each message being a blank canvas, it often falls to the individual author to tell their part of the story in their own way and that naturally leads to differences in style, just as we might find stylistic differences in the code as each author stamps their own identity on it through brace placement and vocabulary. Consequently, we find that different delimiters are used to surround text parameters (or maybe none at all), dates might be formatted locally or in UTC, primitive values output with different units, and spelling & grammar of variable quality too.

This melting pot of text might have some 'character' but it's a royal pain to parse automatically and make sense of at any non-trivial scale. Once you start needing to aggregate and slice-and-dice this data so it can be used to provide overviews of the system health and generate alerts when the proverbial muck hits the fan, then freeform text no longer cuts it – a clear, consistent format needs to be adopted which is machine readable and aggregatable. The alternative is a life spent trying to concoct ever more complex (and unmaintainable) regular expressions to match the dizzying array of similar messages; or not, which is the usual outcome. Machine learning may provide some hope in the future for an otherwise lost cause but until that day comes we need a simpler solution.

Enter stage left – structured logging. Instead of trying to use our natural language, such as English prose, to describe our situation we start to think about it in less emotional terms. Log messages stop being a flat representation of what has passed and instead become richer 'diagnostic' events which can be reacted to programmatically, both within the process as well as outside it [Freeman07]. In its simplest guise instead of writing the traditional message **`File 'Xxx' loaded in 10 ms`** you instead log, say, a **`FileLoaded`** diagnostic event with a property **`DurationInMs`** set to 10 and a **`Path`** property set to the file name, e.g.

```
log.Info($"File '{filename}' loaded in {duration}
ms");
```

now becomes,

```
log.Write(new FileLoaded{ DurationInMs=duration,
Path=filename });
```

As you can imagine, all the usual programming tricks are still available to hide some of the complexity and keep the logging noise down in the code to ensure the functional aspects still shine through, e.g. free functions or extension methods for the most common scenarios:

```
log.FileLoaded(filename, duration);
```

In C# you can leverage the **`using`** mechanism to wrap up the timing and event writing with even less noise thereby giving you some really simple instrumentation [Oldwood13b]:

```
using(journal.FileLoad(filename))
{
  . . .
}
```

(To try and make it clearer that it's not a traditional log file I like to rename the eventing façade something like 'journal' and use terms like 'record' to get away from past traditions.)

Richer diagnostic events

This change in mind-set opens up a new world of possibilities about how you write your monitoring and instrumentation code, and how and where

Chris Oldwood Chris is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days, it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood

With all diagnostic events being richly typed and flowing through a single stream, it's easy to build in in-process feedback loops as well to monitor and react to local conditions

you aggregate it. For example, although modern logging search services are fairly powerful, even if you have used correlation IDs [Oldwood13a] to relate your events within a process, finding and summing them can still be arduous, especially if the entire IT department is sending all their logs to, say, the same Splunk instance. Tools like Splunk allow you to 'join' log event streams (much like a SQL join) but they have limits to keep processing costs down which often means your joins are truncated unless your queries are highly selective. Consequently, you may choose to perform some local aggregation in-process and send out 'summary' events too to ease the burden on the monitoring side. For example, a REST API might generate a summary event that includes the overall timing for handling a request along with a breakdown of the time spent in database calls, outbound to other external services, heavy computations, etc.

With all diagnostic events being richly typed (if you choose to go that far) and flowing through a single stream, it's easy to build in in-process feedback loops as well to monitor and react to local conditions. For example, the Circuit Breaker pattern which Michael Nygard introduced in his seminal book [Nygard07] is a technique for dealing with temporary failures by shutting off the value when things go awry. One way of implementing this is to continuously analyse the diagnostic stream and, if you see a continuous batch of failure events, flick the switch and turn yourself off for a bit. This can be combined with another classic design pattern – Leaky Bucket Counter [Wikipedia] – to slowly turn the tap back on. This elevates the diagnostic event stream from a purely non-functional concern to a more functional one which brings with it greater responsibility. (Logging is often treated as a second-class citizen due to its non-functional nature, but reliable systems elevate non-functional concerns to ensure they get the attention they truly deserve.)

Event structure

Although you can use much richer types to represent each event you need to bear in mind that the goal is to try and aggregate similar kinds of events across multiple processes to get a more holistic feel for what is happening. Consequently, there is still a trade-off between being too loose or too rich and therefore being no better off than we were before. You need just enough structure to ensure a good degree of consistency and to make it easy to add similar classes of events while also ensuring your events are easy to consume in the monitoring system. At its most basic an event might simply be a name and an optional value, e.g.

```
class Event
{
  public string Name { get; }
  public double? Value { get; }
}
```

Naming in software development is always a difficult problem and once again it rears its ugly head here too. You can slice and dice the problem of naming events and their properties however you like but if you want them to be easy to consume in the monitoring tool then you'll want to adopt a scheme that works well with the query facilities on offer. For example, using a namespace style approach which reads narrower from left-to-right, just like code, is a common option, such as `component.operation.duration` for the time taken to perform a specific operation. This could then be translated into a monitoring query like `database.*.duration` to get an overview for the performance of the database. Once you can aggregate key components over a time window you have the opportunity to automatically measure trends in performance which is a commonly missed critical source of system behaviour. (For a short tale about how performance can slowly decrease over time and then suddenly drop off a cliff see [Oldwood14].)

What makes aggregation simpler is to adopt a convention for things which should be aggregated, so for example append `.duration` for timings and, say, `.count` for occurrences of interesting things. This helps reduce (though not eliminate) the chances of the classic mistake of treating '1' as '1 ms' instead of '1 failure' which might make your performance look better or worse than expected. If you're going to be using different units for the same dimension, such as timings, you'll either need to include that as a property or make it clear in the name so that you can correctly sum them, e.g. `.duration-ms` vs `.duration-ns`. With the events now being rendered automatically by code instead of manually formatted by hand you stand a better chance of rescuing the situation if you decide that your precision is too low for one particular kind of event. The Prometheus documentation is a good source of inspiration for categorising and naming events [Prometheus].

If you're wondering about the basic logging attributes you've grown to know and love, such as the timestamp and process ID, well you can use a base class to capture those (for example, see Listing 1).

However, before you go sticking all your favourite extra fields in every event, just ask yourself whether you really need them every time or is that

```
class EventBase
{
  public DateTime Timestamp { get; }
  public string Name { get; }
  public int ProcessId { get; }
}
class DatabaseQueried : EventBase
{
  public int Duration { get; }

  public  DatabaseQueried(string operation
    int duration)
    : this($"database.{operation}.duration")
  {
    Duration = duration;
  }
}
// Client code.
journal.DatabaseQueried("find-customer",
  duration);
```

Listing 1

only a choice you've taken because of the inherent limitations of the traditional method of logging? A fixed width layout was always preferable for a classic log file; maybe it's not so important now?

Human readability

One immediate concern I had when first adopting this approach was that I was so used to reading classic log file narratives that I felt they would be much harder to consume on a development & debugging level rather than a production monitoring basis. For example, I've become used to doing some basic system level testing after making certain kinds of changes to ensure that the non-functional aspects are well represented, i.e. is any new instrumentation as useful as it could be? Given the usual modes of failure do we have recorded what we would generally expect if we have a failure in the new area of functionality? If it involves any I/O, have we timed it so that we can detect problems and visualise trends? This has usually involved reading the event stream manually and playing with classic command line and log viewing tools.

Hence, one question you may have had right from the get-go is about what the event stream looks like outside the process. It depends on the logging infrastructure and destination sinks but the simplest choice might be to just render them as JSON as it has a fairly simple format (less of an 'angle-bracket' tax than XML) and JSON writers are often performant enough for most logging uses. Hence instead of lines of freeform text you might have lines such as this instead:

```
{ Time:"2018-02-16T08:14:01.123",
  Name:"orders.http-error.count", ...}
{ Time:"2018-02-16T08:14:01.123",
  Name:"orders.request.duration-ms", ...}
```

An event stream dumped as raw JSON is clearly not quite as easy to read as plain text. At least, it's perhaps not as easy to read at first, but you soon realise that what really matters is whether you can easily relate the event that you see to where it occurs in the code and what it's telling you about the operation that happened. Surprisingly I got used to this style very quickly; I almost began to feel like Cypher in The Matrix – I stopped seeing the curly braces and quote marks and just saw 'HTTP client error', 'token refreshed', 'database queried', etc. Your mileage will of course vary considerably, but just give it a little time. Apparently, there's more wisdom in the old adage 'omit needless words' than I had quite appreciated.

The other good thing about a decent JSON serializer is that you can usually control the order of serialization, so you can ensure that the most important attributes get written first (e.g. Timestamp and Name) and lesser ones later which will reduce the burden somewhat when eyeballing the log file.

The audience

Another benefit which a more event focused approach gives you is a little more discipline around sticking to true operational events rather than mixing in 'debug' or 'trace' level messages, which are often of little concern to first level support, but which might be useful to a developer doing a full-on post mortem later. This duality of narratives (monitoring vs debugging) is oft debated and what I've personally come to favour is slightly richer events where the 'headline' is aimed at operational use and designed to be aggregated as before but attached to it might be a 'Debug' property with additional context. For example an **`http.request-failure.count`** event might have the response status code as a core property but the response headers and first N bytes of the payload attached to separate **`Debug.Headers`** and **`Debug.Content`** properties should the need arise.

This brings me back to my earlier comment about attaching too much data to every event. Whilst process IDs might be useful to the ops team, thread IDs are probably less so, and therefore consider where and when you really need them. As for deciding in the code what severity an event should be, that probably disappears too as the interpretation shifts from being considered in isolation to being considered at scale, probably in the context of high availability where more transient failures are tolerated. That isn't to say that failing to read a fundamental configuration setting like a database connection string is not pretty serious, but just that failure without recovery is what we're looking for in the short term, not necessarily failure in its own right which is more about trends – let the event state the facts and allow the monitoring to decide if it's noteworthy or not.

Code complexity

There is one other reason I have seen for overly verbose trace-level logging and that is due to overly complex code. When the code is difficult to reason about, and a problem shows up which is difficult to reproduce, it is natural to resort to **`printf`** style debugging by adding extra logging to show which code paths are taken and why. Simple code that is well written and easy to reason about is also easier to diagnose without resorting to such heavy-handed tactics.

Consequently, when writing code we should ask ourselves what we would need to know to be able to determine what significant paths might be taken. If the design does not lend itself to being comprehensible from certain key inputs then we should ask ourselves whether it is too complex and needs to be refactored. Other techniques like Design by Contract may add a little more code here-and-there but this allows us to fail earlier when it's easier to diagnose and therefore can avoid us logging excessively to track the same problem down later when the problem finally manifests itself, e.g. a latent **`NullReferenceException`**.

Improving events

One half of the equation is about having the means in play to represent diagnostic events in a way that can easily be queried and aggregated, the other is about improving the quality of events as you learn more about the system's failure modes. This was rammed home early one Saturday morning when I got a support call about a weekend job failure and the diagnostic message was simply:

```
ERROR: Failed to calibrate currency pair!
```

Tracking down which of the many currency combinations was at fault would have been made so much easier if the author had included in the message the pair of currencies in question that had failed. Whilst debugging the issue I discovered the information was already available at the code site where the message was logged and so I naturally added it right away to help those on support in the future. (Reflecting on and improving exception messages is a related topic I've tackled before [Oldwood15a].)

In an ideal world we would know all the ways that our system can fail and we could code accordingly, but life is just not that fair. Networks like the Internet are awash with services, proxies, caches, load balancers, etc. from different vendors that will behave in different ways when they fail or return an error. Even fairly simple protocols like HTTP are open to different interpretations by different teams, and that's before you bring in the need to work around broken behaviours on supposedly transparent services because they are trying to be 'clever'. The on-premise world does not necessarily fare any better, you still get plenty of bizarre server, network and storage problems to deal with [Oldwood15b].

Consequently, although we can predict certain types of failure, such as a network route being unavailable, we may not know all the ways that it will manifest or what diagnostic information we might need to diagnose it. Waking someone up in the middle of the night every time a network request fails is not going to win you any favours and so we need to analyse our errors and drill in to see if we can categorise them further so that the support team only springs into action when there is actually something to do. During development it's also possible that we can unearth 'impedance mismatches' between various components in our system that on the face of it presents as occasional failures but could lead to significant problems once we reach production scale.

What follows are a couple of examples of how non-catastrophic failures can be diagnosed and either resolved or vastly improved upon to ensure the signal-to-noise ratio remains as high as possible. Each step was a small code change, pushed quickly into production, to gather more information about the situation each time it cropped up. Naturally an approach like continuous delivery really helps to keep the feedback loop short between learning and reacting to the new diagnostic information.

Service misconfiguration

Soon after bringing the initial version of an adapter to bridge the on-premise and cloud-hosted worlds of a service together, the team started seeing occasional errors in the HTTP 5xx range. The general rule is that these are server-side issues and therefore our logic used the classic exponential back-off to ride it out, eventually. What was odd was that the subsequent request always went through fine so whatever the transient condition was it always cleared immediately.

Our first-order approach to diagnostic logging in that component only logged the HTTP status code on failure at this point and so we decided to immediately start capturing more information, such as the HTTP response headers, because this error didn't appear to resonate from the service at the other end. One always needs to be wary of dumping untrusted input even into a logging system and so we were careful to limit what new information we captured.

What this showed was that the HTTP response was coming from Akamai, the company providing some basic security for the underlying service, such as API throttling. Despite only asking for a JSON response we actually got back an HTML content type which raised some interesting questions about our current response handling code. We also decided to include the first half KB of the response content into a 'Debug' section of the logging event so that we could inspect it further when it showed up.

The payload included what appeared to be a standard HTML error response from Akamai that also included an 'error reference' which the support team could look up and see what the underlying issue was. We now felt that we had enough heuristics available to crudely parse this Akamai error response correctly into an event of its own type and expose the error reference directly as a property so that we could quickly look them up with Akamai's own diagnostic tools.

As it turned out, in this instance, the error was due to different idle socket timeouts between Akamai and the AWS load balancer which in itself took some effort to discover. At least we now knew for sure that certain 5xx errors were definitely not from the service and therefore there was no need to go looking elsewhere. Also, unless someone had changed the Akamai configuration, which was very rare, we could say with a high degree of certainty that the error really was transient and out of our control. Consequently, the monitoring dashboard would now show a reduction in generic HTTP error responses in favour of the more specific Akamai ones.

The second example looks at a wider reaching code change in reaction to learning more about the way a system dependency reported certain types of errors.

Error translation

Initially our message handler only had very basic error handling – enough to know to retry on a 5xx HTTP response but to triage a request if it got a 4xx one. In HTTP the 4xx response codes generally mean the client needs to take some corrective action, such as fixing its request format (400), authenticating itself (401), or checking if the right resource was being requested (404). We never saw a 404 but we did see occasional 400s which was worrying as we didn't know what we could have done wrong to create a malformed request for such a simple query.

Luckily, by adding a little extra diagnostic data, we discovered the upstream service provided a richer error payload, in JSON, that we could attach to a `Debug` property of the, then simple, `http.client-error.count` event, along with any response headers, just like earlier. From doing this we discovered that the 'service specific' error code was tantamount to a 'not found' error, which you'd usually report with a 404 HTTP status code. The service was only providing a simple lookup and therefore we could attach the identifier we were looking up in our request onto a new, richer diagnostic event designed to show which IDs were unfortunately not correctly mapped.

This not only enhanced the monitoring side of the equation, but it also meant that we could use a much richer exception type to convey this unusual condition to the outer layers of our code. It turned out that these missing mappings were never going to be addressed any time soon and therefore pushing the request onto a 'slow retry' queue was a complete waste of time and we might as well drop them on the floor until further notice.

Aside from a reduced number of 'poisoned' messages to deal with on our side, by including the information directly in the monitoring data this also meant the other team which owned the upstream service could query our logs and find out how bad the problem was and whether it was getting better or worse over time. We could also easily see if any particular IDs cropped up regularly and ask if they could be fixed manually. A better choice of HTTP status code in the first place would have made life slightly simpler but we would have still needed to understand the problem to ensure that we removed the unnecessary noise and reacted accordingly; either way we couldn't have left it as it was.

Summary

The freeform text log file has been with us for a long time and still continues to serve us well in the right situations. However, as the size and complexity of our systems grows we need something more 'grown-up' that can sensibly cater for the needs of the operations team as well as those developing it (even if they are one and the same). These two forces are often at odds with each other, not out of malice but out of ignorance and so we should redress the balance by putting their diagnostic needs on an equal par with our own. Their hands are effectively tied and so those who work with the code need to be the ones to cater for them as well.

Moving towards a more structured approach to logging brings with it an improvement in monitoring as we start to consider what each diagnostic event means, not in isolation but in the context of the system as a whole. By making it easy to aggregate related events across time, by component, or by subsystem we can get a more holistic feel for what is going on under the hood. Then, taking these same data points over a much longer period of time we can get a feel for what the trends are in the system as it grows so that we are not caught off guard when those cyclic peaks come around second or third time.

Finally, we also need to accept that "there are more things in heaven and earth, Horatio, than are dreamt of in your philosophy". We don't know all the ways our system will fail but by continuing to invest in refining the way we capture and report errors we stand a better chance of ensuring the signal stands out from the noise. ■

References

[Freeman07] 'Test Smell: Logging is also a feature', Steve Freeman, http://www.mockobjects.com/2007/04/test-smell-logging-is-also-feature.html

[Nygard07] *Release It! Design and Deploy Production-Ready Software*, Michael T. Nygard, https://pragprog.com/book/mnee/release-it

[Oldwood13a] 'Causality – Relating Distributed Diagnostic Contexts', Chris Oldwood, *Overload* #114, https://accu.org/index.php/journals/1870

[Oldwood13b] 'Simple Instrumentation', Chris Oldwood, *Overload* #116, https://accu.org/index.php/journals/1843

[Oldwood14] 'Single Points of Failure – The SAN', Chris Oldwood, http://chrisoldwood.blogspot.co.uk/2014/09/single-points-of-failure-san.html

[Oldwood15a] 'Terse Exception Messages', Chris Oldwood, *Overload* #127, https://accu.org/index.php/journals/2110

[Oldwood15b] 'The Cost of Not Designing the Database Schema', Chris Oldwood, http://chrisoldwood.blogspot.co.uk/2015/12/the-cost-of-not-designing-database.html

[Prometheus] 'Metric and Label Naming', Prometheus, https://prometheus.io/docs/practices/naming

[Wikipedia] 'Leaky bucket', Wikipedia, https://en.wikipedia.org/wiki/Leaky_bucket

The Interface to Component Pattern and DynaMix

Dynamic Polymorphism is hard in C++. Borislav Stanimirov demonstrates how the DynaMix library helps.

Much of the evolution of modern language design has been in improving static polymorphism through better features for generic and metaprogramming. Popular languages such as Java, C#, and especially C++ have been going through a renaissance of sorts in this regard for the past 10 or so years. On the other hand, languages with good metaprogramming support such as D have been enjoying renewed interest and popularity. Newer languages, such as Nim, are being developed where metaprogramming is the central focus. Dynamic polymorphism on the other hand has been left in the background. For C++, the C++11 standard added **`std::function`** and **`std::bind`**, but almost no improvements have been added to the language to support dynamic polymorphism in an object-oriented context (**`final`** and **`override`** being the minor exceptions to this).

Object oriented programming doesn't include dynamic polymorphism in its formal definition, but in practice it has come to imply it. In many contexts (such as Java) the inverse is also true. Given the bad publicity OOP has been getting through the years, it is no surprise it has been living as a side note in languages such as C++ which are oriented towards maximal performance and type safety. Indeed many C++ programmers have forgotten, or deliberately chosen to forget, that C++ is, among other things, an object-oriented language.

While some of the criticism of OOP is concerned with performance, most of it is focused on how bad particular implementations (for example Java's) are at accomplishing certain complex business requirements. It is the opinion of the author that this is not a problem with OOP as a whole. OOP with dynamic polymorphism is often a great way to express business requirements. Highly dynamic languages such as Python, Ruby, JavaScript, and many more, are thriving in fields dominated by business logic. It is no wonder that many pieces of software are implemented with a core in some high-performance language such as C or C++ and business logic modules in a dynamic and more flexible language (lua being an especially popular choice for games, for example).

Besides the support for better OOP features, such an approach has other benefits, like the possibility to 'hot-swap' live code while the program is running and in some cases, using their strong DSL-creation mechanisms, to delegate some code to non-programmers. Unfortunately there are drawbacks, too.

The performance is inevitably worse. Even with JIT, save for very few edge cases, interpreted code is slower than compiled and optimized C++. With JIT 2-5 times slower is the rule but in other, JIT-unfriendly edge cases, ten or more times slower is not unexpected. Without JIT some languages perform even worse. For a Ruby program to be hundreds of times slower than its C++ counterpart is a common occurrence.

Borislav Stanimirov Borislav has been a C++ programmer for 15 years. In the past 11 he has been programming video games. He has worked on C++ software for all kinds of platforms: desktops, mobile devices, servers, and embedded. His main interests are software architecture and design, and programming languages

There needs to be a binding layer between the core and the business logic language. This is a piece of code (often of considerable size) which has no other purpose than to be a language bridge. It adds more complexity to a project and a lot of time needs to be invested in developing and maintaining it.

There is functionality duplication. Even with the best of intentions, it's often highly impractical to call small utility functions through the binding layer. As a result, many such functions have an implementation in both the core and the business logic language. Thousands of lines of duplicated functionality is a normal occurrence in such projects, which is often times the source of duplicated bugs.

It is evident that if the aforementioned drawbacks are prohibitive for a project, some kind of new approach is needed for better OOP support in a high-performance language.

Even though developers of libraries in high-performance languages have been largely ignoring OOP functionalities, there are still some efforts in improving them. For C++, the most notable developments gaining popularity recently are polymorphic type-erasure wrappers. These include the somewhat ancient Boost.TypeErasure [Boost] and the much more modern Dyno [Dyno], and Facebook's Folly.Poly [Facebook]. They offer major improvements of the vanilla C++ OOP polymorphism. They allow for better separation of interface and implementation. They are non-intrusive (no inheritance needed). They are more extensible since you define interfaces and classes separately. In some cases they can potentially be faster, but in any case they are not slower than virtual functions.

However… they are more or less the same in terms of architecture. There still are interface types and implementation types. They offer great improvements of OOP polymorphism in C++ but they are not much better than Java or C# in terms of how you design the software. They are just not compelling enough to ditch scripting languages.

One of the most popular OOP techniques in dynamic languages is to compose and mutate objects at runtime. Ruby offers a very concise and readable way of accomplishing this, so consider the piece of code in Listing 1 (overleaf) – part of the gameplay code of an imaginary game (gameplay means business logic in game-dev jargon).

Those are Ruby mixins. Note that in C++ circles the term mixin exists [ThinkBottomUp] and it's used for something similar. It's a way of composing objects out of building blocks, but at compile time through CRTP. Now with the Interface to Component pattern a similar functionality can be accomplished in C++ and any other language which has at least Java-like OOP support.

The pattern is based on composition over inheritance (much like almost every fix of OOP-specific problems). Here is an annotated C++ implementation using Interface to Component of the same gameplay (see Listing 2, overleaf).

Interface to Component is being widely used in pieces of software with complex business logic such as CAD systems, some enterprise software, and games. It is especially popular in mobile games because their target

```
module FlyingCreature
  def move_to(target)
    puts "#{self.name} flying to #{target.name}"
  end
  def can_move_to?(target)
    true # flying creatures can move anywhere
  end
end

module WalkingCreature
  def move_to(target)
    puts "#{self.name} walking to #{target.name}"
  end
  def can_move_to?(target)
    # walking creatures cannot walk over obstacles

!self.world.has_obstacles_between?(self.position,
  target.position)
  end
end

# composing objects

hero = GameObject.new
hero.extend(WalkingCreature)
hero.extend(KeyboardControl)
  # controlled by keyboard
objects << hero # add to objects

dragon = GameObject.new
dragon.extend(FlyingCreature)
dragon.extend(EnemyAI) # controlled by enemy AI
objects << dragon # add to objects

main_loop_iteration # possibly the hero can't move

# give wings to hero
hero.extend(FlyingCreature) # overrides
WalkingCreature's methods
main_loop_iteration # all fly

# mind control dragon
dragon.extend(FriendAI)
  # overrides EnemyAI's methods
main_loop_iteration # dragon is a friend
```

Listing 1

```
class Component // base to all components
{
public:
  virtual ~Component() {}
protected:
  friend class GameObject;
  // pointer to owning object
  GameObject* const self = nullptr;
};

// component interface for movement
class Movement : public Component
{
public:
  virtual void moveTo(const Point& t) = 0;
  virtual bool canMoveTo(const Point& t)
    const = 0;
};
```

Listing 2

```
// component interface for Control
class Control : public Component
{
public:
  virtual const Point& decideTarget() const = 0;
};

// Main object
class GameObject
{
  // component data
  std::unique_ptr<Movement> _movement;
  std::unique_ptr<Control> _control;
  // ... other components

  void addComponent(Component& c) {
    const_cast<GameObject*>(c.self) = this;
  }
public:
  void setMovement(Movement* m) {
    addComponent(*m);
    _movement.reset(m);
  }
  Movement* getMovement() {
    return _movement.get();
  }

  void setControl(Control* c) {
    addComponent(*c);
    _control.reset(c);
  }
  Control* getControl() {
    return _control.get();
  }
  // ...
  // GameObject-specific data
  const Point& position() const;
  const World& world() const;
  const std::string& name() const;
  // ...
};

// component implementations
class WalkingCreature : public Movement
{
public:
  virtual void moveTo(const Point& t) override {
    cout << self->name() << " walking to " << t
         << "\n";
  }
  virtual bool canMoveTo(const Point& t)
      const override {
    return
      !self->world().hasObstaclesBetween(
         self->position(), t);
  }
};
class FlyingCreature : public Movement
{
  virtual void moveTo(const Point& t) override {
    cout << self->name() << " flying to " << t
         << "\n";
  }
  virtual bool canMoveTo(const Point& t)
      const override {
    return true;
  }
};
```

Listing 2 (cont'd)

```
  // composing objects
  auto hero = new GameObject;
  hero->setMovement(new WalkingCreature);
  hero->setControl(new KeyboardControl);
  objects.emplace_back(hero);

  auto dragon = new GameObject;
  dragon->setMovement(new FlyingCreature);
  dragon->setControl(new EnemyAI);
  objects.emplace_back(dragon);

  mainLoopIteration();
  // possibly the hero can't move

  // give hero wings
  hero->setMovement(new FlyingCreature);
    // overriding WalkingCreature
  mainLoopIteration();
    // all characters fly

  // mind-control dragon
  dragon->setControl(new FriendAI);
    // overriding EnemyAI
  mainLoopIteration();
    // the dragon is a friend now
```

Listing 2 (cont'd)

hardware is less powerful than PCs which makes the developers less likely to sacrifice performance for an additional dynamic language. In can be (and often is) combined with the ENTITY-COMPONENT-SYSTEM pattern [Wikipedia] so some components are updated in their own systems appropriately, while others serve as polymorphic implementers of object-specific functionalities. It is a pattern which is easy to understand and relatively easy to implement and modify according to specific needs. For example to have multicast support, one only has to make a vector of components from a given interface. Unfortunately Interface to Components comes with its own set of drawbacks.

The object is a coupling focal point. Every component interface needs to be declared inside (or worse, included with naïve implementations like the one from the example above). In C++ frequent changes to components and object structure will change the object class and trigger a recompilation of the entire business logic system in a project. *This will be mitigated once we have modules, but they are not here yet.*

Most notably though, interfaces are limiting. Imagine the following addition to the Ruby example from above (Listing 1):

```
module AfraidOfSnow
  def can_move_to?(target)
    self.world.terrain_at(target) != Terrain::Snow
  end
end

dragon.extend(AfraidOfSnow)
main_loop_iteration # dragon won't fly to snow
```

We added a mixin which overrides one of the methods of the movement interface. There is simply no easy way to accomplish this with Interface to Component. We could inherit from flying creatures but it is not only flying creatures who could be afraid of snow. This override is applicable for every type of movement. We could try solving this [Afanasiev16] with the aforementioned CRTP mixins, but this will put a lot of code in template classes and lead to horrible compilation times, and even if we fix this with explicit instantiations, we're left with the problem of having to know what we override. The only solution is to split the interface into 'movement method' and 'movement availabilty'... until we're left with a huge code base of single-method interfaces and the burden of having to deal with knowing which to add or remove in different scenarios.

DynaMix [DynaMix-a] is a C++ library which solves these problems. Its name means Dynamic Mixins as it is for dynamic polymorphism what CRTP mixins are for static polymorphism. It allows the users to compose and mutate 'live' objects at runtime and offers a big amount of additional features which may be needed in the development of the project. It was conceived and developed back in 2007 as a proprietary library in a PC MMORPG project, and it was reimplemented and open-sourced in 2013. It has since been used in several mobile games by different teams and companies.

Listing 3 is an annotated implementation of the same gameplay, this time using DynaMix.

```
// declare messages
// DynaMix doesn't use class-interfaces. Instead
// the interface is provided through messages,
// which are declared with macros like this.
// A message is a standalone function which some
// mixins may implement through methods
DYNAMIX_MESSAGE_1(void, moveTo,
  const Point&, target);
DYNAMIX_CONST_MESSAGE_1(bool, canMoveTo,
  const Point&, target);

// define some mixin classes
class WalkingCreature
{
public:
  void moveTo(const Point& t) {
   // `dm_this` is a pointer to the owning object
   // much like `self` was in our previous
   // examples.
   // Note that due to the fact that C++ doesn't
   // have unified call syntax, we cannot write
   // code like dm_this->name(). Instead messages
   // are functions where the first argument is
   // the object.
    cout << name(dm_this) << " walking to "
         << t << "\n";
  }
  bool canMoveTo(const Point& t) const {
    return
      !world(dm_this).hasObstaclesBetween
      (position(dm_this), t);
  }
};
class FlyingCreature
{
public:
  void moveTo(const Point& t) {
    cout << name(dm_this) << " flying to " << t
         << "\n";
  }
  bool canMoveTo(const Point& t) const {
    return true;
  }
};
// define mixins
// The mixin definition macros "tell" the library
// what mixins there are and what messages they
// implement
DYNAMIX_DEFINE_MIXIN(WalkingCreature,
  moveTo_msg & canMoveTo_msg);
DYNAMIX_DEFINE_MIXIN(FlyingCreature,
  moveTo_msg & canMoveTo_msg);
  // compose objects
  auto hero = new dynamix::object;
  dynamix::mutate(hero)
    .add<WalkingCreature>()
    .add<KeyboardControl>();
  objects.emplace_back(hero);
```

Listing 3

```
  auto dragon = new dynamix::object;
  dynamix::mutate(dragon)
    .add<FlyingCreature>()
    .add<EnemyAI>();
  objects.emplace_back(dragon);

  mainLoopIteration();
    // possibly the hero can't move

  // Replace WalkingCreature with FlyingCreature
  dynamix::mutate(hero)
    .remove<WalkingCreature>()
    .add<FlyingCreature>();
  mainLoopIteration(); // all objects fly

  // Replace EnemyAI with FriendAI
  dynamix::mutate(dragon)
    .remove<EnemyAI>()
    .add<FriendAI>();
  mainLoopIteration(); // the dragon is friendly
```

Listing 3 (cont'd)

Now, this seems like a poorer implementation than the one we created for the Interface to Component example. Namely it seems that the user needs to know what mixin is already in the object in order to change the functionality which is already inside. This is just the case for this simple example. Let's move on to the `AfraidOfSnow` feature (Listing 4).

DynaMix is a free and open-source library under the MIT license. Its source [DynaMix-a] and the documentation [DynaMix-b] are available.

In short, what does the library do? It provides the type `dynamix::object`, an empty bag of sorts, whose instances can be extended by classes (or mixins) which the users have written. Extending the objects with mixins provides them with the functionality of those mixins.

```
class AfraidOfSnow
{
public:
  bool canMoveTo(const Point& t) const {
    return world(dm_this).terrainAt(t)
      != Terrain::Snow;
  }
};
// Here we define the mixin and set a priority to
// the message. This tells the library that when
// this mixin is added to an object which already
// implements the message with a lower priority
// (the default being 0) this implementation must
// override the existing one.
DYNAMIX_DEFINE_MIXIN(AfraidOfSnow,
  priority(1, canMoveTo_msg));

  // overriding FlyingCreature::canMoveTo
  dynamix::mutate(dragon)
    .add<AfraidOfSnow>();
  mainLoopIteration();
    // the dragon cannot fly to snow

  // restoring previous functionality
  // A feature which was not available in the
  // Interface to Component implementation and
  // even not possible with Ruby's mixins
  dynamix::mutate(dragon)
    .remove<AfraidOfSnow>();
  mainLoopIteration();
    // the dragon can fly freely again
```

Listing 4

The term 'message' is inspired by Smalltak's strict differentiation between a message and a method (although in many OOP languages 'message' has fallen out of favor). Basically a message is the interface, while the method is the implementation. The differentiation is important in a late binding context such as the one in DynaMix. As you can see, with the library users can create messages with the message macros. Those macros generate a standalone function with `dynamix::object` as the first argument. They also generate some functions which the library will use to register the message and fill a 'virtual table' for the object which implement it. As we said, this is a late binding scenario, so an empty object implements no messages. A runtime error will be triggered if you call a message for an object which doesn't implement it.

The mixins are the building blocks of DynaMix objects. A mixin is a class created by a library user, and registered with the mixin registration macros. Those macros instantiate internal data structures for the library which associate the mixin with the messages it implements so they can be added to the virtual call table of an object when the user adds this mixin to it. When users mutate objects by adding mixins, the library instantiates them internally using their default constructors (custom allocators can be provided for cases where the memory block or constructor call needs to be user defined). Thus an object instance has a collection of unique mixin instances within it. Again, it's composition over inheritance – the universal tool. Mixin instances are allocated and constructed or deallocated and destroyed when the object is mutated. Thus a mixin instance cannot be shared between object instances.

Each unique combination of mixins creates an internal object type in the library. It holds the virtual call table for this mixin combination. Objects hold a pointer to their type. The message function which is generated by the message macros finds the appropriate method pointer in the call table of the object, then finds the mixin pointer within the object, and performs the call. This (as any dynamic dispatch) requires some dereferencing indirections but it's always *O(1)*. The object types save a lot of memory per object, since typically there are thousands or even millions of objects but tens or hundreds of unique types. They also mean that the order with which you add mixins doesn't matter. Adding `a` and then `b` will produce the same type with the same virtual call table as adding `b` and then `a`. This is a notable difference between DynaMix and the typical Interface to Component implementation, which has its version of a virtual call table per object and the order of mutation matters. In DynaMix determining which method overrides which doesn't happen with the order of mutation but with message priorities.

For more implementation details and a full list of features you can check out the code [DynaMix-a] or the docs [DynaMix-b]. ■

References

[Afanasiev16] 'Combining Static and Dynamic Polymorphism with C++ Mixin classes' https://michael-afanasiev.github.io/2016/08/03/Combining-Static-and-Dynamic-Polymorphism-with-C++-Template-Mixins.html

[Boost] Type.Erasure: http://www.boost.org/doc/libs/1_65_1/doc/html/boost_typeerasure.html

[DynaMix-a] Source: https://github.com/iboB/dynamix

[DynaMix-b] Documentation: https://ibob.github.io/dynamix/

[Dyno] Dyno: https://github.com/ldionne/dyno

[Facebook] folly.Poly: https://github.com/facebook/folly/blob/master/folly/docs/Poly.md

[ThinkBottomUp] 'C++ Mixins – Reuse through inheritance is good... when done the right way' http://www.thinkbottomup.com.au/site/blog/C%20%20_Mixins_-_Reuse_through_inheritance_is_good

[Wikipedia] Entity–component–system: https://en.wikipedia.org/wiki/Entity–component–system

5 Reasons NOT to Use std::ostream for Human-Readable Output

C++'s ostream can be hard to use. Sergey Ignatchenko suggests we use the {fmt} library instead.

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

This is NOT yet another printf-vs-cout debate

First of all, to avoid being beaten really hard, I have to say that I am perfectly aware of all the arguments presented in favour of 30+-year-old `std::ostream` (that is, compared to `printf()` which arguably comes from 50+-years-old BCPL) – and moreover, that I am NOT going to argue for `printf()` in this article.

The arguments usually used to push `cout` over `printf`, are the following [C++ FAQ]:

1. `iostream` is type-safe.

   'No Bugs' comment: I am the last person to argue about this one.
2. it is less error-prone (referring to reducing redundancy)

   'No Bugs' comment: while saying that reducing redundancy is the same as being less error-prone is a bit of stretch in general (in quite a few cases, redundancy is exactly what keeps us from making silly mistakes), in the context of the `cout`-vs-`printf()` debate, I can agree with it.
3. it is extensible (allowing you to specify your own classes to be printed).

   'No Bugs' comment: very nice to have indeed.
4. `std::ostream` and `std::istream` are inheritable, which means you can have other user-defined things that look and act like streams, yet that do whatever strange and wonderful things you want.

   'No Bugs' comment: TBH, I fail to see why being inheritable is an advantage *per se*; especially as extending existing functionality doesn't depend on inheritance (at least, as long as no virtual functions are involved, and I don't see many of them in `std::ostream` as such). The best I can make out of this one is understanding it as 'being able to provide my own underlying streambuf to be used by `ostream`', which does qualify as an advantage (at least over `printf()`,which doesn't provide such an option at all: more on this below).

Sergey Ignatchenko has 20+ years of industry experience, including being an architect of a stock exchange, and the sole architect of a game with hundreds of thousands of simultaneous players. He currently writes for a software blog (http://ithare.com), and translates from the Lapine language a 9-volume book series 'Development and Deployment of Multiplayer Online Games'. Sergey can be contacted at sergey.ignatchenko@ithare.com

Once again, I am NOT going to argue with the points above (doing so would certainly start another ~~World~~ Flame War); instead, I just want to take them as a starting point (clarifying the one which isn't obvious to me, so it is specific enough for our purposes).

ostream is far from perfect

Even with all its advantages over 50+year-old `printf()`, `ostream` is still far from perfect – at least for human-readable outputs.

Ok, so far we have seen the good side of `ostream`; however (conspicuously omitted from [C++ FAQ]), it has quite a few downsides too, especially if we concentrate on specific use cases for `std::ostream`. A whole bunch of very popular use cases for `ostream`s involve formatting output which is intended to be read by human beings. Two popular examples of such formatting include:

- Formatting output which is shown to the end-user (usually in some kind of UI, whether graphical or not).
- Formatting output which is sent to text-based logs (which tends to apply both to the Client-Side and to the Server-Side).

  Note that, strictly speaking, Server-Side text-based logs can be divided into (a) text logs used for monitoring purposes, and (b) text logs for post-mortem analysis, with a recent movement towards making (a) structured rather than free-text based. Still, I am sure that (b) is there to stay as free-text based, so the text logging use case will still stand even if the movement towards structured logging for monitoring purposes succeeds.

As it said on the tin, we're going to concentrate on output intended for human beings – and while we're at it, we'll keep in mind the two major use cases above. And, as I am going to present a point of view which – while it was articulated previously in [Moria] and [NoBugs] – is certainly not as popular as the four points above (yet?), I am going to be significantly more verbose than [C++ FAQ].

So, in no particular order, here they are: the major drawbacks of `ostream`s when used to format human-readable outputs.

Drawback number 1: i18n

"Vantage number one!" said the Bi-Coloured-Python-Rock-Snake. "You couldn't have done that with a mere-smear nose. Try and eat a little now."

~ Bi-Coloured-Python-Rock-Snake from *Just So Stories*

The first major problem with using `ostream`-like chevron-based formatting for human-readable strings is internationalization. Let's take a look at a piece of code which formats a simple message for the UI of an online poker game:

```
some_ostream << winner.name << " shows "
  <<   winner.cards << " and wins $"
  << pot_size / 100 << "." << std::setw(2)
  << std::setfill('0') << pot_size % 100;
  // we have pot_size stored in cents, but have to
  // display it in a more conventional manner
```

If there is a need to use a different order of parameters in the translated version, this can easily be done by the translator without any involvement from developers

NB: for our purposes, let's skip the discussion about localizing currency signs and dots-vs-commas; in particular, for online games, the former happens to be not a question of locality, but a question of what currency this site really uses, and nobody gives a damn about the latter.

When trying to translate this code, it happens to suffer from two huuuuge (actually, bordering on insurmountable) problems, namely:

- Translations NEVER work by translating isolated words. In other words, there is no point in asking a translator to translate a fragment such as "shows" into a different language. Such translations (even if translators are silly enough to do them) will never work, simply because for translations context is everything – but with the code above, the context is buried within C++ code, and is not easily extractable (we DON'T want to teach translators C++, do we?)
- Moreover, the order of the parameters we want to substitute (the `winner.name`, `winner.cards`, and `pot_size`) can be *different* in a human-readable language other than English; with the code above, *this would mean that potentially we have to rewrite the code for each target human language* (ok, for 3 parameters, we can say that there aren't more than 3!=6 possible combinations we have to code, but IMNSHO it is still 6x too much).

Now, let's come to specific examples; to illustrate better than `ostream` alternatives throughout this article, I (by definition) have to use something different from `ostream`. However, as I don't want to use `printf()` for this purpose (to make it even more clear that I am NOT advocating a return to `printf()`) I'll use one of Python's format options (the curly braced one) to illustrate how things can be done. In Python, our formatting looks as follows:

```
print("{0} shows {1} and wins ${2}.{3:02d}"
.format(winner.name,winner.cards,pot_size/100,
pot_size%100))
```

Here, we have our string (with placeholders in curly brackets) and can easily pass it to the translation team . While we will still have to replace our original literal with something read from a file at runtime, it is still nothing compared to the need to rewrite the whole `ostream`-based thing (with all the possible variations for the order of parameters). Most importantly, with Python-like formatting, both our i18-related points above are addressed:

- Our original phrase to be translated exists as a self-contained literal. As practice shows, these tend to be perfectly translatable (in some cases, comments about the meaning of {0}, {1}, and {2} may need to be added to help translators better understand the context – but that's about it, and most real-world phrases are already more or less self-contained).
- If there is a need to use a different order of parameters in the translated version, this can *easily* be done by the translator without any involvement from developers (which, BTW is exactly the way it should be).

Drawback number 2: multithreading

"Vantage number two!" said the Bi-Coloured-Python-Rock-Snake.
"You couldn't have done that with a mere-smear nose.
Don't you think the sun is very hot here?"
~ Bi-Coloured-Python-Rock-Snake from *Just So Stories*

While i18n is mostly in the realm of strings intended for some kind of UI, our second drawback is mostly related to logging in a multithreaded environment.

NB: for this drawback, I'll use different example code – which is more typical for logging than for formatting for a UI, and that's where this particular problem is more likely to manifest itself.

If you have ever written innocent-looking code such as

```
logging_stream << "Event #" << std::setw(8)
<< std::setfiller('0') << std::hex << event_id
<< ": a=" << std::dec << a << " b=" << b << "\n";
```

and then tried to run it in two different threads *simultaneously*, you know that the code above can easily generate all kinds of weird outputs, including such beauties as

```
Event #Event #0089a1b2c3d4e5f6: a=12: a= b=
b=345678
<\n>
<\n>
```

In addition to being completely unreadable, there is absolutely no way to figure out how digits from '345678' were distributed between one `a` and two `b`s coming from different threads (and in which order BTW).

The reason for it is simple: with `ostream`, instead of calling one implementation function, we're calling *several separated* `<<` operators; in turn, this means that the largest possible synchronization unit for `cout` stream is not a *phrase* (~= "one line we want to output"), but merely each of the items between `<<` chevrons. This inevitably leads to potentially having outputs such as the one above.

Sure, somebody can say "Hey, you should place a mutex lock above that line" – and it would help; however, placing such mutex locks is not just error-prone, but error-prone-squared because (a) it is easy to forget to place it, and (b) it is even easier to forget to unlock it *right after the* `cout` *line* (which, in turn, can easily lead to a *huuuuge* performance degradation for no reason whatsoever).

A better alternative is proposed in [P0053R7], where a special temporary object (an instance of `class osyncstream`, which is derived from `ostream`) is constructed on top of our real `ostream` object (such as `cout`). Then, the `osyncstream` object will buffer all the output written to it via `<<` operators, and will write to the underlying `cout` only at the point of being destructed. This ensures that all the output written to `osyncstream` is guaranteed to be written in one piece <phew />. IMO, `osyncstream` is indeed a pretty good workaround for this particular problem (at any rate, *much* better than mutexes), but it still has the following significant issues:

(a) unless we limit ourselves to one-line uses of our `osyncstream` object (more precisely, to creating an `osyncstream` instance only

it is easy to forget to limit the scope of our osyncstream, which can lead to reordering of whole ‘phrases’ in our log

temporarily), writing to the underlying stream in the destructor becomes rather counterintuitive, and it is easy to forget to limit the scope of our **`osyncstream`**, which can lead to reordering of whole ‘phrases’ in our log (it won’t look as bad as reordering of the words shown above, but can still cause significant confusion when reading the logs);

(b) extra buffering won’t come for free (especially as the current proposal seems to use allocations <ouch !/>); and

(c) [P0053R7] won’t help with the other issues discussed in the article (though maybe it might help to deal with our next drawback – sticky flags – too).

Drawback number 3: sticky flags

Anyone who has tried to do some formatting which goes beyond the textbook using cout has encountered a huuuge problem that

> *With* **`ostream`***, formatting modifiers (such as hex-vs-dec, filler, etc.) are considered an attribute of the stream, not of the output operation.*

In other words: formatting flags, once applied, ‘stick’ to the stream. This, in turn, means that if you forget to revert them back, you’ll obtain an unexpectedly formatted output (and of course, it won’t be noticed until production, and will manifest itself in exactly the place where it causes the maximum possible damage).

This problem becomes especially bad in scenarios where we have one global stream (such as **`cout`** or a log file). *In fact, it means that our formatting flags become a part of the GLOBAL mutable program state* – and last time I checked, everybody of sane mind (including those people who are arguing for **`cout`**), agrees that global mutable state is a Bad Thing™.

In fact, this problem is so bad, that Boost even has a special class to deal with it! With Boost’s **`ios_flags_saver`**, our code will look like:

```
boost::io::ios_flags_saver ifs(logging_stream);
logging_stream << "Event #" << std::setw(8)
  << std::setfiller('0') << std::hex << event_id
  << ": a=" << std::dec << a << " b=" << b
  << "\n";
```

However, even with such an RAII-based workaround, once again it is error-prone: it is easy to forget to add the **`ios_flags_saver`** – especially if the policy is to use it only when some sticky manipulators are applied (and if our project Guidelines say ‘always use **`ios_flags_saver`**’, it would be a violation of the ‘not paying for what we don’t use’ principle, and would still be rather error-prone).

Drawback number 4: readability

“Vantage number three!” said the Bi-Coloured-Python-Rock-Snake.
“You couldn’t have done that with a mere-smear nose.
Now how do you feel about being spanked again?”
~ Bi-Coloured-Python-Rock-Snake from *Just So Stories*

```
// UI formatting
// guard is probably NOT required here, as we’re
not
// likely to work with UI strings from multiple
// threads
boost::io::ios_flags_saver ifs(some_ostream);
some_ostream << winner.name << " shows "
  << winner.cards << " and wins $"
  << pot_size / 100 << "." << std::setw(2)
  << std::setfill('0') << pot_size % 100;

//logging
std::lock_guard<std::mutex>
  guard(logging_stream_mutex);
boost::io::ios_flags_saver ifs(logging_stream);
logging_stream << "Event #" << std::setw(8)
  << std::setfiller('0') << std::hex << event_id
  << ": a=" << std::dec << a << " b="
  << b << "\n";
guard.unlock();//as discussed above, we don’t
want
// to keep lock longer than really necessary
```

Listing 1

Now, let’s try to write down our full examples of formatting human-readable output using **`ostream`** (while keeping all the considerations above in mind). To summarize, our rather simple formatting code examples will look like Listing 1.

When looking at the code in Listing 1, I cannot help but think that it ~~has been spanked by the Elephant’s Child~~ has fallen from the Ugly Tree™ (hitting all the ugly branches on the way down). And whenever somebody tells me that this code is *readable*, I can only ask them to compare it with the way the same thing is done in pretty much all other languages but C++ (yes, even in C – though using an unmentionable function); in particular, in Python it would look like Listing 2.

Formally speaking, the **`ostream`**-based code above has between 2x and 4x more characters, and between 2.5x and 5x more non-whitespace YACC tokens, than the demonstrated format-string based alternative, and

```
  //UI formatting
  print("{0} shows {1} and wins ${2}.{3:02d}"
  .format(winner.name,winner.cards,pot_size/100,
    pot_size%100))

  //Logging
  print("Event #{0:08x}: a={1:d} b={2:d}"
   .format(event_id, a, b))
```

Listing 2

there is **more than one library** out there which not only has **all the advantages of ostream over printf()** but also **fixes all the drawbacks** of the ostream we listed

while brevity does not necessarily equate to better readability, in the case of a 300–400% overhead, it usually does.

And if looking at it informally, with just (hopefully) an unbiased programmer's eyes:

> *I think the answer to 'which of the two pieces of code above can be seen as readable' is very obvious*

(hint: I do NOT think that the **ostream**-based one qualifies as such).

Drawback number 4.5: writing customized underlying stream could be better

Yet another drawback of the **ostream** (BTW, this one stands regardless of whether it is being used for human-readable output) is that the process of writing the underlying stream is rather non-obvious and is seriously error-prone. I don't want to go into details here (it is way too long since the last time I did it myself) but [Tomaszewski] describes what I remember pretty well, including observations such as "Properly deriving from **std::streambuf** is not easy and intuitive because its interface is complicated", and making "a very subtle bug which took me several hours to detect".

To be perfectly honest, it is still MUCH better than not being unable to write a customized stream *at all* (as is the case for **printf()**), but – as I noted above – I am not speaking in terms of **printf()**, and being prone to subtle bugs is certainly not a good thing for those who need to rewrite an underlying **streambuf**.

Drawback number 5: something MUCH better exists

All the musing about the drawbacks of **ostream** would remain a rather pointless ranting if not for one thing: a library *exists* which has *all* the **ostream**-like advantages listed in [C++ FAQ], and *none* of the drawbacks listed above.

Actually, there are several such libraries (Boost format, FastFormat, tinyformat, {fmt}, and FollyFormat – and probably something else which I have forgotten to mention). I have to note that, personally, I don't really care too much *which* one of the competing new-generation format libraries makes it into the standard (except, probably, for Boost format, which is way too resource-intensive when compared to the alternatives). In general, I (alongside with a very significant portion of the C++ community) just want *some* standard and better-than-**iostream** way of formatting human-readable data.

Out of such newer formatting libraries I happen to know {fmt} by Victor Zverovich the best, *and* it certainly looks very good, satisfying *all* the points from C++ FAQ, and avoiding *all* the **iostream** problems listed above. As {fmt} is also the only new-generation library with an active WG21 proposal [P0645R1], it is the one I'm currently keeping my fingers crossed for. (NB: in the past, there was another proposal, [N3506], but it looks pretty much abandoned).

In this article, I am not going to go into lengthy discussion about {fmt} vs the alternatives – but will just mention that with {fmt}, our examples will look like:

```
fmt::print("{0} shows {1} and wins ${2}.{3:02d}",
  winner.name,winner.cards,pot_size/100,
  pot_size%100);
  //this is C++, folks!

  fmt::print("Event #{0:08x}: a={1:d} b={2:d}",
    event_id, a, b);
```

This alone allows us to avoid most of the problems listed above (and FWIW, I'd argue it is even more readable than Python); in addition, {fmt} is type-safe, extensible, supports both **ostream** and **FILE*** as underlying streams (with the ability to add your own stream easily), beats **ostream** performance-wise, et cetera, et cetera.

C++ Developer Community on formatting approaches

After all the theorizing about different formatting approaches, let's see what real-world developers are saying about the different libraries available for this purpose. First, I have to note that even before the advent of the new generation of format libraries – *and in spite of enormous pressure exerted by quite a few C++ committee members via their numerous publications in favour of* **cout** – real-world C++ developers were badly split on the question "what is better – **cout** or **printf()**" (see, for example, statistics in [StackOverflow] and [Quora]). Now, with {fmt} available, developers *seem* to agree that it is the best real-world option out there [Reddit]; just two quotes from *top-upvoted* comments (which prove nothing, of course, but do count as anecdotal evidence):

- I am already using **{fmt}** all over my projects but having it in the **std** would be great.
- So happy this is steadily transitioning in **std**. One of the best formatting (and i/o) libs out there overall. Even without the localization argument, I've always found iostreams to be less convenient.

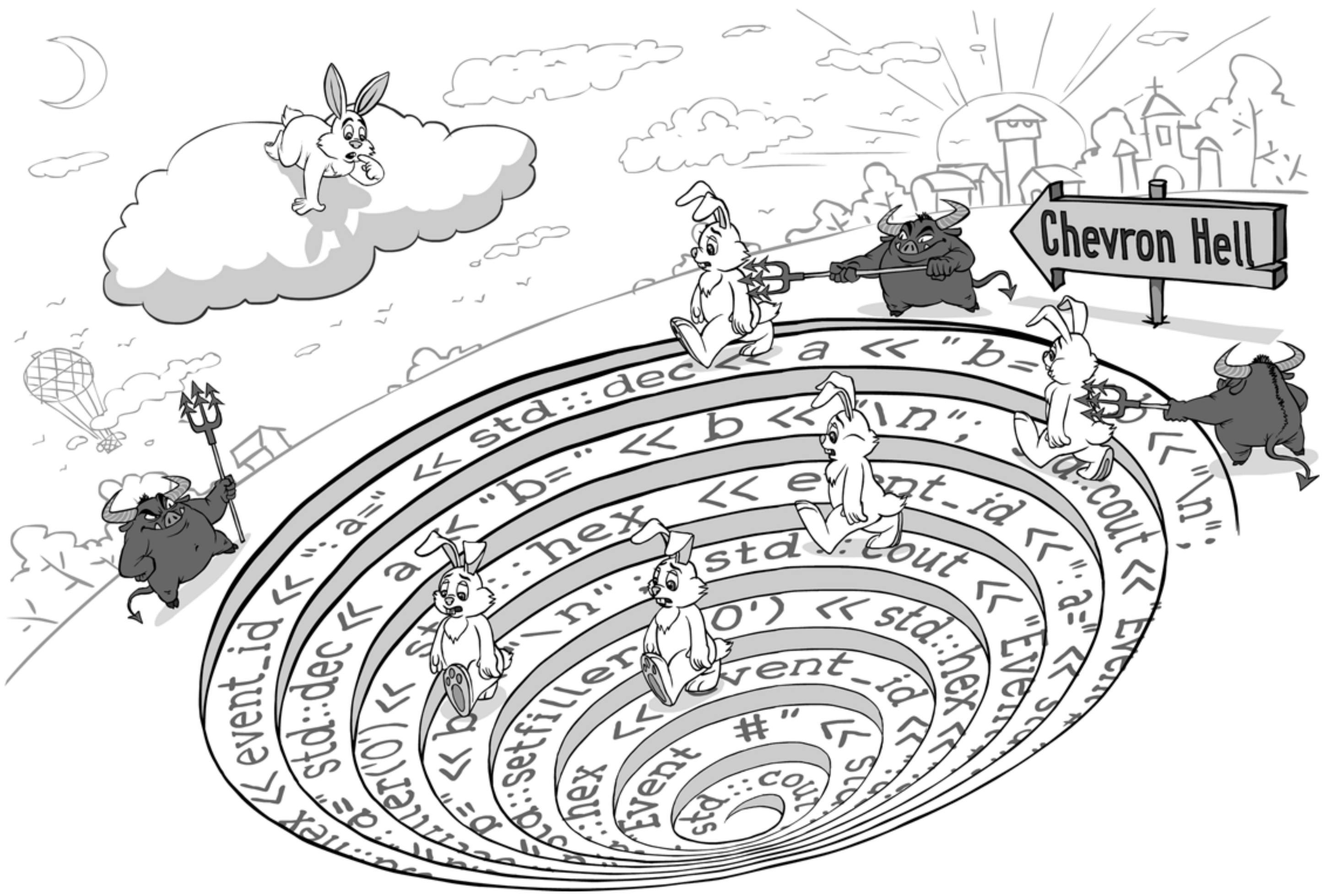
Yes, I know it sounds like a bad commercial, but I am pretty sure these comments are genuine.

Oh, and if somebody in WG21 still has any doubts about what-C++ developers want to use for human-readable formatting, please let me know: I'll organize a survey to get more formal numbers.

Conclusion

We took a look at **std::ostream** and issues with its real-world usage when formatting output intended for human beings. As a side note, we observed that most of the problems with **std::ostream** in this context arise from it working as a stream (either char stream, or word/token stream) while human beings tend to communicate in phrases or sentences, and one thing **std::ostream** is badly lacking is support for those phrases/sentences so ubiquitous in the real world.

Moreover, as we noted, there is more than one library out there which not only has all the advantages of **ostream** over **printf()** but also fixes *all* the drawbacks of the **ostream** we listed above. IMNSHO, there is no question of 'what is better to use' (that is, for human-readable outputs). This means that our (= 'real-world C++ developers') course of action is very clear:



- Start using {fmt} as much as possible (well, you may choose some other library over {fmt}, but IMO fragmentation is a bad thing for such a library, so unless you have some very specific requirements, I suggest using {fmt} as a *de facto* standard). Aside from the direct benefits we'll get from using it, it might help to iron out any subtle issues left (such as 'how to implement *compile-time* type safety'), and to make the proposal to WG21 more solid.
- Keep our fingers crossed hoping that WG21 *will* take the P0645R1 proposal into the standard (though with the pace of changes making through WG21, I will have to pray really hard that it happens before I retire <sad-wink />). ■

References

[C++ FAQ] C++ FAQ, https://isocpp.org/wiki/faq/input-output#iostream-vs-stdio

[fmt] A modern formatting library, https://github.com/fmtlib/fmt

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

[Moria] IOStream Is Hopelessly Broken, https://www.moria.us/articles/iostream-is-hopelessly-broken/

[N3506] Zhihao Yuan, A printf-like Interface for the Streams Library

[NoBugs] 'No Bugs' Hare, #CPPCON2017. Day 1. Hope to get something-better-than-chevron-hell, http://ithare.com/cppcon2017-day-1-hope-to-get-something-better-than-chevrone-hell/

[P0053R7] Lawrence Crowl, Peter Sommerlad, Nicolai Josuttis, Pablo Halpern, C++ Synchronized Buffered Ostream, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0053r7.pdf

[P0645R1] Victor Zverovich, Lee Howes. Text Formatting. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0645r1.html

[Quora] When would you use fprintf instead of cerr/iostream in C++?, https://www.quora.com/When-would-you-use-fprintf-instead-of-cerr-iostream-in-C++

[Reddit] A Chance to Get Readable Formatting: {fmt}, https://www.reddit.com/r/cpp/comments/72krvy/a_chance_to_get_readable_formatting_fmt/

[StackOverflow] 'printf' vs. 'cout' in C++, https://stackoverflow.com/questions/2872543/printf-vs-cout-in-c

[Tomaszewski] Krzysztof Tomaszewski, Deriving from std::streambuf, https://artofcode.wordpress.com/2010/12/12/deriving-from-stdstreambuf/

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague

Practical Cryptographical Theory for Programmers

Cryptography is a daunting subject.
Deák Ferenc helps you get started.

Throughout the written history of humankind, the need to hide information has been omnipresent in almost all aspects of our lives. From safely transmitting messages to our fighting troops in ancient times to the simplest click we execute on our browser today to log in to our favourite social media site, there is an abundance of information that is being transmitted between participants who wish their information to be safe, secure and available only to them.

The most common way of achieving this goal is to use some form of encryption scheme, which takes an existing message and – using a series of operations – transforms the message into a scrambled form that cannot be read until a corresponding, but inverse, operation called decryption is applied to it, which will reveal the original message.

In this article, we will focus on a beginner level introduction to cryptography that is applicable to our everyday tasks. This requires the encrypting and decrypting of messages as a form of safe communication between two (or more) participants in a communication channel. There will be a short discussion on cryptography in order to have a better understanding and overview of the terminology we use, and we will provide practical examples on how to use cryptographic functions in our code.

In order to not to scare the reader away and also to keep the size under a certain digestible limit, the article intentionally skips the advanced cryptographical terminologies, and we will not dive into the deep mathematical foundations that the theory of cryptopgraphy is built upon.

There are several excellent books written about cryptography by renowned cryprographers whose prestigious work has greatly contributed to the advancements in the field, so we do not even try to condense all that information here, but just give a generic overview of what we should know about cryptography at a level where we can start using it in our daily work.

The article can be read either by starting from the beginning and reading through the terminologies part followed then by the code, or the other way around, by starting directly with the code and referencing backward into the terminologies part when something unknown pops up.

And, of course, if you become interested in cryptography after reading this article I always recommend reading more and more material in this field and even attend a dedicated training course, since this article can be just a 'teaser' into this huge field. It is impossible to cover everything while still keeping it readable, not forgetting to mention that the physical size of this journal cannot compete with the size of a book, so don't be afraid to do extra research and invest extra time into deepening your knowledge.

The Ultimate goal of Cryptography

There are three main goals in cryptography that can be viewed as the holy grail:

1. **C** stands for Confidentiality: Ensuring that only authorized parties are able to understand the encrypted data.
2. **I** stands for Integrity: Ensures that only authorized parties can modify the data and to ensure that the data that has left the sender is the same as the data received by the recipient.
3. **A** stands for Authentication: Ensures that anyone who supplies or accesses sensitive data is an authorized party.

And since we used the **Authorized** term frequently, here is a loose definition for it:

> anyone who has a particular secret and has permissions (usually obtained directly from the source of the plaintext) or more mundanely put: knows the password, so he can log in and has proper rights to perform operations in the system (read, write, execute, access, ...).

There is a difference between **Authenticated** and **Authorized** where authenticated refers to someone who is verified to be whom he or she is supposed to be and Authorized we've seen before. Or more mundanely: Authenticated knows the password and is allowed log in with provided credentials.

And last, but not least: the integrity aspect comes closely tied to the non-repudiation facet of a message interchange, ie.: if you have sent it, you cannot deny that you have sent it.

Terminology in cryptography

Before we start with the practical (read: writing code) part of the article, there is a need for a very short introduction presenting a few definitions in order to have a brief understanding of the terms used in the field. Without this introduction it would be more difficult to understand the example code.

Plaintext

Plaintext is nothing else but the message we wish to transform using cryptographical algorithms in order to protect the information it stores. Plaintext is usually easily interpretable by humans using various techniques, such as reading its content.

Ciphertext

Ciphertext is the resulting data of a cryptographical algorithm when it is applied to a plaintext. It is supposed to be unreadable by humans or machines and only by using the correct algorithm should the originating plaintext be revealed.

XOR

Since the bitwise operation `XOR` (Exclusive OR) is mentioned several times in the article, just a quick reminder that XOR is the logical operation that outputs true (1) only for different inputs.

Deák Ferenc Ferenc has wanted to be a better programmer for the last 15 years. Right now he tries to accomplish this goal by working at FARA (Trondheim, Norway) as a system programmer, and in his free time, by exploring the hidden corners of the C++ language in search for new quests. fritzone@gmail.com

A Cryptanalyst is highly skilled in the dark art of breaking code (or just has access to a multi-billion-dollar super computer doing brute force attacks)

The following is the truth table of the XOR operator.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

The cipher

According the the Oxford dictionary a cipher is: (NOUN) ‘A secret or disguised way of writing’. For us programmers, the cipher is just another word for a pair of encryption/decryption algorithms.

- **Encryption**, as per the definition, is the process of converting the **plaintext** into the unreadable code known as **ciphertext** in order to prevent unauthorized access to it.
- **Decryption** is the inverse operation of the encryption.

The encryption algorithm uses a key to scramble the data, which can be viewed as your secret password (but it’s actually more), and the decryption algorithm must use also a key (same or different) to retrieve the plaintext.

In cryptography there are two mainstream cipher types utilized today:

1. symmetric ciphers are algorithms that use the same secret key for encryption and decryption of the data.
2. asymmetric ciphers are algorithms that use a pair of keys for the encrypting and decrypting process.

For anyone interested, [Golodetz08] has an excellent description on the inner works of probably the most well known public key algorithm (RSA) so for this article we will be focusing on symmetric ciphers only, and from now on any reference to a cipher in this article must be interpreted as ‘symmetric cipher’.

And finally, the definition of a **Secret Key** is: a piece of information, which is used to encrypt and decrypt messages in a cipher. It is confusable with **password**; however, it should not be due to the following major differences:

- a **password** is created by a ‘user’ by choosing a secret, but it is rarely used in properly set up systems to directly encrypt/decrypt data due to being considered ‘cryptographically weak’ (humans tend to choose data that they are familiar with when choosing a password, such as dictionary words, pet names, birth dates, or just simply ‘password’).
- a **secret key** is data which is the result of an algorithm applied to a **password** which gives ‘cryptographically strong’ data that can be safely used in the algorithms.

Attack of the Ciphers (and how to defend ourselves)

Since most of the encryption activities happen with a very specific purpose (ie: hide something the enemy is not supposed to know), in the adversary camp there is usually someone with the specialized role of **Cryptanalyst** who tries to obtain the secret information. Cryptanalysts are highly skilled in the dark art of breaking code (or just have access to a multi-billion-dollar super computer doing brute force attacks) but the worst of all is that they have access to our ciphertext. They can manipulate the ciphertext in order to obtain the plaintext and to derive the method used to encrypt which will allow them to decrypt other messages too and even reveal the secret key.

Types of attack

Several types of attack have been devised during the history of cryptography. We will present shortly a few (but not all), because when implementing cryptography in a system, it is wise to know what behaviour to expect from someone who tries to break your system.

Ciphertext-only attack

In this case, the cryptanalyst has access to a set of ciphertexts, and his ultimate goal is to retrieve the plaintext. A possible attack scenario is that the cipher was chosen with a small key space, thus via brute force the attacker can try all the possible keys. For example, DES (Data Encryption Standard) has keys of 56 bits, which are easily broken using modern technologies [DESCRACK]. Attacks on the ciphers used in GSM technology (A5/1 and A5/2) are also ciphertext-only attacks when intercepted message streams from phone conversations can be decrypted using dedicated solutions.

Chosen plaintext attack

In this scenario, the attackers can obtain the ciphertexts for chosen plaintexts. By analyzing the result ciphertext, they can gain information regarding the security of the encryption cipher and the algorithm it is using. In this case, the attacker has access to a ‘black box’ which generates ciphertext from individual plaintexts.

Chosen ciphertext attack

For this attack, the attacker can obtain the decrypted form of chosen ciphertexts. By analyzing it, information regarding the key can be obtained. As in the previous case, the attacker has access to a ‘black box’ which it will query with the chosen encrypted sequences.

Known plaintext attack

In this scenario, the attacker has access to both the plaintext and the ciphertext. These two can be used to reveal further information, such as encryption keys or algorithms.

Hardening your ciphertext

To make the life of the Cryptanalyst harder, extra protection steps can be taken to obtain a more secure ciphertext. These involve introducing a **Salt** (which is just a sequence of random bytes) to the encryption algorithm (more specifically, the password is ‘salted’ with it) which among other benefits, makes the usage of 'rainbow tables' (which are precomputed tables usually for obtaining the hashes of passwords) impossible.

Another element used in the encryption/decryption process is the **Initialization vector** which, similarly to the salt, is also a sequence of random numbers and is used in the initialization phase of the encryption algorithm in order to prevent the same plaintext generating the same ciphertext when the same algorithm is applied to it.

And the last element which will make our encryption safer is a **Nonce** which is just a plain number (coming from 'number used once') again used (only once) in order to make different ciphertext for the same input data.

The salt and the initialization vector are not considered private information, thus it is widely accepted to have them being sent over communication channels.

Types of ciphers

Currently two mainstream types of symmetric ciphers are in use:

1. Block Ciphers
2. Stream Ciphers

A **Block Cipher** is a deterministic pair of algorithms which operates on fixed-length groups of bits, which are called a block. One of the algorithms is used for encryption, the other one for decryption (which in mathematical terms is defined to be the inverse function of the encryption).

The algorithms have two inputs: a block (size: **N** bits) and a key (size: **K** bits). Both algorithms return an output block (size: **N** bits).

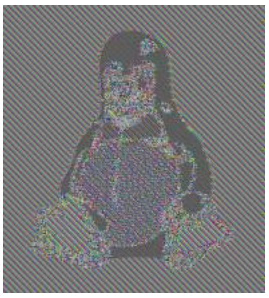
For a detailed description of the mode of operation of a block cipher, please consult [BlockCipher]. And in order to keep this article in a digestible size, we will focus our attention on block ciphers.

For the sake of brevity, let's just mention that a **Stream Cipher** is a symmetric key algorithm (the same key is used for encryption and decryption) where the bits of the plaintext are combined (practically XOR-ed) with the bits of a pseudorandom cipher stream (called keystream).

Block cipher modes of operation

By definition, a block cipher operates on fixed length blocks of data, so the first operation that is done by the algorithm is the splitting of the plaintext into blocks of the required size and then each block is encrypted independently. This mode (called the **ECB** – Electronic Codebook) has the disadvantage that equal plaintext block will always generate the same ciphertext block.

You always should avoid using ECB while performing encryption, here is a proof of everyone's favourite penguin image encrypted with ECB [ECB_TUX]:



In order to overcome this limitation, several algorithms have been designed that use randomization of the plaintext using an additional value (such as the **Initialization vector** mentioned earlier) to obtain a different ciphertext for identical plaintext.

The most commonly used of these modes are:

- **CBC**: Cipher Block Chaining – In this mode, the current block of plaintext is combined (using XOR) with the previous ciphertext block before being encrypted. The first block is combined with the initialization vector.
- **PCBC**: Propagating Cipher Block Chaining – In this mode, the current block of plaintext is combined (using XOR) with both the previous plaintext block and the previous ciphertext block before being encrypted. The first block is combined with the initialization vector.
- **CTR**: Counter – This mode of operation acts like a stream cipher. It generates the next keystream block by encrypting successive values of a 'counter' which can be generated by using a nonce and combining it with a nonrepetitive value generated by a function. Usually an increment by 1 of the value is the simplest operation.

A very detailed description of these is presented at [BlockCipherModes].

Padding

Since we cannot always expect the length of a message to be the exact multiple of the size of the block the algorithm operates on, some modes of operation (CBC, for example) require that the last block is padded with bytes of various origin. For this article, we will stick to the PKCS#7 padding mode presented in [RFC5652], which pads the input data with a number of **D** = **N** - **K** bytes, with their value being exactly **D** [PKCS7].

Hashing

Hashing is a method of ensuring the integrity (one of the presented goals) of data by applying a hash function to it and retrieving the associated hash value. A hash function is a function which maps the data (of arbitrary length) to a data of fixed size in a deterministic manner (ie: same input data will always yield the same output value). It should be impossible to retrieve the input data using only the hash value. For purposes of cryptographic needs, cryptographic hash function are used which are widely presented in [CRHASH]. The output of the hash function is often called a digest.

For this article we will be focusing our attention to SHA-256, however this does not block anyone from experimenting with other functions.

SHA-256

As presented in [SHA256], SHA-2 is a set of cryptographic hash functions designed by the United States National Security Agency (NSA) from which we use the one which provides a digest of 256 bits. It is implemented and used in lots of applications and protocols, and as per 2018 it is considered safe for widespread use.

MAC or HMAC

The term MAC stand for Message Authentication Code, which is basically just a way to confirm the integrity of a message with a given key. Since these functions are usually constructed using a hashing function, the term HMAC (Hash-based MAC) is also used. What this does is nothing else than to calculate a cryptographically secure hash of a given data combined with a given secret key [HMAC].

Key derivation functions

In Cryptography, a **K**ey **D**erivation **F**unction is a scheme which takes an initial key and, through a series of operations, derives a cryptographically secure, uniformly distributed, strong secret key which can be used in cryptography operations.

For this article we will focus on PBKDF2 [PBKDF2], which is fairly modern and secure implementation of this function.

PBKDF2 derives a key of a specified length from a password, using a randomly generated salt, through a number of iterations on the basis that 'more is better' and, since 'more is slower', this is also a practical defense against brute force attacks.

AES

AES in cryptographical terms stands for 'Advanced Encryption Standard' is also known as 'Rijndael' and was developed by two Belgian cryptographers, Vincent Rijmen and Joan Daemen. The cipher was adopted worldwide for encryption of electronic data after it won the competition of U.S. NIST (National Institute of Standards and Technology) in 2001.

AES is a fast (to the level that modern microprocessors include a dedicated set of instructions [AES-NI]), safe and secure algorithm according to the cryptographical community. Also, Bruce Schneier [Schneier00], who developed Twofish (a competitor for Rijndael) for the same NIST competition, acknowledged:

> I do not believe that anyone will ever discover an attack that will allow someone to read Rijndael traffic.

AES works on blocks with a fixed block size of 128 bits and a key size of 128, 192, or 256 bits, which makes it secure enough for all required cryptographical needs of modern systems.

A full specification of the algorithm is presented in [AES] and [AES-NIST], and a more detailed overview is presented in *Cryptography: Theory and Practice* [Stinson05] chapter 3.6.1 (page 105) so anyone interested can follow up there after reading the article.

For the moment we should just know that, in the practical part of the article, we will use AES for encryption and decryption.

Practicing cryptography

In the 'practical' part of the article, we will focus on a real-life scenario, where a hypothetical application created in javascript needs to send messages to an imaginary server, which was written in C++.

Please note: the code presented has just the purpose of being an example, it is definitely not to be put into production as it is. It is intentionally kept simple and readable, and it is upon the future developer to expand it in order to reach production grade.

In order to keep the article compact in size, we will not introduce several topics here that are out of scope for this article. We just assume that there is some agreement between the parties on how to securely transmit the password between the two endpoints, as you can see right now it is hardcoded into the source file. You definitely should NOT do this in production code: use a proper key management system for this.

For now, we just pretend that there is a proper protocol for sending the message between the two endpoints; again, for the example, we just 'copy/pasted' the encrypted message into the decryption code. This is definitely not a real-life situation.

The client-side libraries

For javascript, there are several libraries available (you can take a look at: [ClientCryptLibs]) which perform the required encryption/decryption operations on user data; for example, cryptico or cryptojs.

Following the documentation, **crypto-js** ([CryptoJS]) is easy to set up and use, but this should not hinder anyone trying out any other libraries.

The server-side library

On the server side, we can choose from a wide variety of libraries as per [ServCryptoLibs]. For demonstration purposes, I decided to use **Botan** [Botan] since it's written in a fairly modern C++ dialect and it has implemented a huge variety of standard, safe and even not so well known algorithms.

The Botan site has an excellent 'getting started' section, which covers the build steps and has a wide selection of examples which can be instantly taken over into your code.

Another nice feature I appreciated with Botan is that it can create an amalgamation build thus enabling you to effectively include the source of the entire library (or just required parts of it, since the build tool is highly configurable) into your project to not to have to worry about libraries, linking and missing dependencies.

Client-side code

Please note, that in order to properly run the code presented in this section in a browser, I had to set up a local web server serving a static HTML page from my local file system, which had all the necessary HTML syntax … but this is out of the scope for this article. (Appendix A will present the full HTML page.)

```
var iterations = 1000;
var keySize = 256;
function encrypt (msg, pass)
{
  var salt
    = CryptoJS.lib.WordArray.random(128 / 8);

  var key = CryptoJS.PBKDF2(pass, salt, {
    keySize: keySize / 32,
    iterations: iterations,
    hasher: CryptoJS.algo.SHA256
  });

  var iv
    = CryptoJS.lib.WordArray.random(128 / 8);

  var encrypted = CryptoJS.AES.encrypt(msg,
    key, {
      iv: iv,
      padding: CryptoJS.pad.Pkcs7,
      mode: CryptoJS.mode.CBC
    });

var hash = CryptoJS.HmacSHA256(msg, key);
var hashInBase64 =
  CryptoJS.enc.Base64.stringify(hash);
var result = hashInBase64.toString() + "_"
  + salt.toString()+ iv.toString()
  + encrypted.toString();
  return result;
}
var encrypted
  = encrypt("Hello World", "S3cr3tP4sw");
window.alert(encrypted);
```

Listing 1

The javascript code is in Listing 1.

Let's step through it.

```
var iterations = 1000;
```

`iterations` is the number of iterations which will be used by PBKDF2 in order to generate the key from our 'master' key (the **`pass`** parameter) which is practically used in encrypting the data (**`msg`**). There are various recommendations about the size of this number, but all of them agree that it should be a big one.

For the sake of the demonstration I chose it to be 1000; however, for real life situations a much bigger number is recommended.

```
var keySize = 256;
```

Tells us that we will attempt to use a key size of 256 bits; this is what we send to the **`PBKDF2`** call.

```
var salt
  = CryptoJS.lib.WordArray.random(128 / 8);
```

The line will create a random salt, using the **`random`** CryptoJS function, of length 16 to be used together with the password in the **`PBKDF2`** call below.

```
var key = CryptoJS.PBKDF2(pass, salt, {
  keySize: keySize / 32,
  iterations: iterations
});
```

This line is the one which actually creates the key that is used in the encryption. The ingoing parameters are the password we received as parameter, the salt we have generated, the size of the key we expect back and the iterations we want to spend on generating the key. The `keySize` parameter is the size of the key in **words** where a word on today's architectures is typically 32 bits.

By default, the CryptoJS implementation

```
var iv = CryptoJS.lib.WordArray.random(128 / 8);
```

will create another random sequence of 16 bytes with the role of initialization vector that will be used in the encryption phase.

```
var encrypted = CryptoJS.AES.encrypt(msg, key, {
  iv: iv,
  padding: CryptoJS.pad.Pkcs7,
  mode: CryptoJS.mode.CBC
});
```

This is the actual encryption step. Here we see how everything comes together when we are trying to encrypt the message with the key that was generated from a (theoretically weak) password, and using the initialization vector, specifying the padding (PKCS7, presented in a previous paragraph) and the operation mode (CBC) we also discussed before.

```
var hash = CryptoJS.HmacSHA256(msg, key);
var hashInBase64
  = CryptoJS.enc.Base64.stringify(hash);
```

These lines calculate an HMAC for the given message and the key.For the purposes of message integrity, it is recommended to supply the MAC for the message in order to be able to verify whether someone has tampered with it or not.

```
var result = hashInBase64.toString() + "_"
  + salt.toString()+ iv.toString()
  + encrypted.toString();
```

This is the line which calculates the result by simply concatenating the MAC, a separator (_), the salt, the initialization vector (remember, these are not considered private information) and the encrypted message. Finally we return the result, obtaining for example the following string:

```
Fo/7rjwOjHUO0iK/REOpl4uq4L+12zA4tfc/YnNLeTg=
_be9df31d0005ebff75c68790f7730100fc588ee586cee4cc
777327d2a010c4a1Pww/3i54DbH77FHr3+SJyg==
```

This can be decomposed into:

- the HMAC of the message: `Fo/7rjwOjHUO0iK/REOpl4uq4L+12zA4tfc/YnNLeTg=`
- the salt = `be9df31d0005ebff75c68790f7730100`
- the initialization vector = `fc588ee586cee4cc777327d2a010c4a1`
- the actually encrypted data = `Pww/3i54DbH77FHr3+SJyg==`

Please note the followings:

- since salt and iv are random, this method will return a different string every time
- `_` can be used as a separator, because the B64 alphabet does not contain this symbol.

Server side code

With Botan, creating a decrypter is just a few lines of code, should not be more than Listing 2.

```
std::string decrypt(const std::string& encrypted,
  const std::string& password,
  const std::vector<uint8_t>& salt,
  const std::vector<uint8_t>& iv,
  std::size_t iterations,
  const std::string& expected_mac)
try
{
  Botan::PKCS5_PBKDF2 pbkdf2(new Botan::HMAC(
    new Botan::SHA_256));
  Botan::SymmetricKey key(pbkdf2.derive_key(32,
    password, &salt[0], salt.size(),
    iterations).bits_of());
  Botan::InitializationVector the_iv(iv.data(),
    iv.size());
  Botan::Pipe pipe(new Botan::Base64_Decoder,
    Botan::get_cipher("AES-256/CBC/PKCS7", key,
    the_iv, Botan::DECRYPTION));
  pipe.process_msg(encrypted);
  std::string result = pipe.read_all_as_string();
  Botan::Pipe mac_pipe(
    new Botan::MAC_Filter("HMAC(SHA-256)", key),
    new Botan::Base64_Encoder);
  mac_pipe.process_msg(result);
  std::string mac_result
    = mac_pipe.read_all_as_string(0);
  if (mac_result != expected_mac)
  {
    return "";
  }
  return result;
}
catch (const std::exception& )
{
  return "";
}
```

Listing 2

In Appendix B, we will present the full C++ source that uses the output from the cryptojs source and decodes the text fully, but for now let’s examine this snippet line by line.

```
std::string decrypt(const std::string& encrypted,
  const std::string& password,
  const std::vector<uint8_t>& salt,
  const std::vector<uint8_t>& iv,
  std::size_t iterations)
```

is just the declaration of the method: it expects all necessary input data to be sent in. Since some Botan functions might throw `std::exception` derived exceptions, I have found improved readability for this specific purpose by packing the body of the function into a function-try-block, hence the try. Certainly, if you wish to fine-grain your error reporting, you always can have several `try-catch` blocks on the various steps.

```
Botan::PKCS5_PBKDF2 pbkdf2(new Botan::HMAC(
  new Botan::SHA_256));
```

will create the PBKDF2 object that we will use at a later stage to derive the master key from the provided password. Please note that the hasher method obviously has to match the one which was used in creating the encrypted text in the javascript code `CryptoJS.algo.SHA256` or the decryption will fail. Botan takes care of the dynamically allocated object, by storing it in a `std::unique_ptr`.

```
Botan::SymmetricKey key(pbkdf2.derive_key(32,
  password,
  &salt[0],
  salt.size(),
  iterations
  ).bits_of()
);
```

This line creates the key which will be used in the decryption of the data. Again, the number of iterations and the size of the key must match the one that we have used in the javascript code.

```
Botan::InitializationVector the_iv(iv.data(),
  iv.size());
```

Creates an initialization vector object Botan can work with from the data we have procided.

```
Botan::Pipe pipe(new Botan::Base64_Decoder,
  Botan::get_cipher("AES-256/CBC/PKCS7",
    key,
    the_iv,
    Botan::DECRYPTION
  )
);
```

A Botan pipe is very similar to the notion of pipe that exists in many operating systems. Data comes in at the beginning, goes through various steps and comes out at the end. For our needs, we require a

Botan::Base64_Decoder object, since cryptojs provided B64 encoded data, and the output of this object (Botan calls them filters) will go into a Cipher object, obtained via:

```
Botan::get_cipher("AES-256/CBC/PKCS7",
  key,
  the_iv,
  Botan::DECRYPTION)
```

The syntax is straightforward: we ask Botan to provide a cipher for decryption (**Botan::DECRYPTION**) for the given key and initialization vector. We would like to use **AES-256**, with operation mode **CBC** and padding **PKCS7**.

The Botan pipe will own these objects so we don't need to worry about freeing them at a later stage.

When we have the pipe set up, we simply ask it to process our message:

```
pipe.process_msg(encrypted);
```

and finally retrieve the result as a string:

```
std::string result = pipe.read_all_as_string();
```

Now comes the verification of the integrity of the message:

```
Botan::Pipe mac_pipe(
  new Botan::MAC_Filter("HMAC(SHA-256)", key),
  new Botan::Base64_Encoder);
mac_pipe.process_msg(result);
std::string mac_result
  = mac_pipe.read_all_as_string(0);
```

Will create another botan pipe in order to calculate the MAC of the message with the key.

```
if (mac_result != expected_mac)
{
  return "";
}
```

And these lines simply verify that the MAC we have received as part of the message matches with the one we have calculated from the decrypted message and the key.

And that's it. ■

Appendix A

My web server is set up in a way that all the required javascript files (cryptojs) are to be found inside the `js` folder in the root of the page, however you can set it up any way you desire, and you even can use online CDN sites to load cryptojs files. See Listing 3.

Appendix B

I have used **unhex** from **boost::algorithm** in order to convert a hex string into its corresponding binary vector. If you don't have experience with (or access to) boost algorithms feel free to use any other mechanism that will achieve the same results.

Splitting up the incoming string in a much more programmatical manner than presented in Listing 4 is left as an exercise for the reader.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">

  <title>hypothetical client</title>
  <script type="text/javascript" src="js/aes.js"></script>
  <script type="text/javascript" src="js/hmac.js"></script>
  <script type="text/javascript" src="js/pbkdf2.js"></script>
  <script type="text/javascript" src="js/sha256.js"></script>

  <script type="text/javascript">

  var iterations = 1000;
  var keySize = 256;
  function encrypt (msg, pass)
  {
    var salt = CryptoJS.lib.WordArray.random(16);

    var key = CryptoJS.PBKDF2(pass, salt, {
      keySize: keySize/32,
      iterations: iterations,
      hasher: CryptoJS.algo.SHA256
      });

    var iv = CryptoJS.lib.WordArray.random(16);

    var encrypted = CryptoJS.AES.encrypt(msg, key, {
      iv: iv,
      padding: CryptoJS.pad.Pkcs7,
      mode: CryptoJS.mode.CBC

    });

    var hash = CryptoJS.HmacSHA256(msg, key);
    var hashInBase64 = CryptoJS.enc.Base64.stringify(hash);

    var result = hashInBase64.toString() + "_"
      + salt.toString()+ iv.toString()
      + encrypted.toString();
    return result;
  }
  var encrypted = encrypt("Hello World", "S3cr3tP4sw");
  window.alert(encrypted);

  </script>
</head>
<body>
</body>
</html>
```

Listing 3

References

[AES]: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[AES-NI]: https://en.wikipedia.org/wiki/AES_instruction_set

[AES-NIST]: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[BlockCipher]: https://en.wikipedia.org/wiki/Block_cipher

[BlockCipherModes]: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

[Botan]: https://botan.randombit.net/

[ClientCryptLibs]: https://github.com/gabrielizalo/JavaScript-Crypto-Libraries

[CRHASH]: https://en.wikipedia.org/wiki/Cryptographic_hash_function

[CryptoJS]: https://github.com/brix/crypto-js

[DESCRACK]: https://en.wikipedia.org/wiki/EFF_DES_cracker

[ECB_TUX]: http://en.wikipedia.org/wiki/Image:Tux_ecb.jpg This image is derived from File:Tux.jpg, owned by Larry Ewing (lewing@isc.tamu.edu) and created using The GIMP (https://www.gimp.org/)

[Golodetz08]: Stuart Golodetz 'RSA Made Simple', *Overload*, June 2008

[HMAC]: https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

[PBKDF2]: https://en.wikipedia.org/wiki/PBKDF2

[PKCS7]: https://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS7

[RFC5652]: https://tools.ietf.org/html/rfc5652#section-6.3

[Schneier00]: https://www.schneier.com/crypto-gram/archives/2000/1015.html

```
#include <botan/key_filt.h>
#include <botan/aes.h>
#include <botan/pbkdf2.h>
#include <botan/hmac.h>
#include <botan/pipe.h>
#include <botan/sha2_32.h>
#include <botan/b64_filt.h>
#include <botan/filters.h>

#include <boost/algorithm/hex.hpp>

#include <string>
#include <vector>
#include <iostream>

std::string decrypt(const std::string& encrypted,
                 const std::string& password,
                 const std::vector<uint8_t>& salt,
                 const std::vector<uint8_t>& iv,
                 std::size_t iterations,
                 const std::string& expected_mac)
try
{
  Botan::PKCS5_PBKDF2 pbkdf2(new Botan::HMAC(
    new Botan::SHA_256));
  Botan::SymmetricKey key(pbkdf2.derive_key(32, password,
    &salt[0], salt.size(), iterations).bits_of());
  Botan::InitializationVector the_iv(iv.data(), iv.size());
  Botan::Pipe pipe(new Botan::Base64_Decoder,
    Botan::get_cipher("AES-256/CBC/PKCS7", key, the_iv,
    Botan::DECRYPTION));
  pipe.process_msg(encrypted);
  std::string result = pipe.read_all_as_string();
  Botan::Pipe mac_pipe(new Botan::MAC_Filter(
    "HMAC(SHA-256)", key), new Botan::Base64_Encoder);
  mac_pipe.process_msg(result);
  std::string mac_result = mac_pipe.read_all_as_string(0);
  if (mac_result != expected_mac)
  {
    return "";
  }
  return result;
}
catch (const std::exception& )
{
  return "";
}
std::vector<uint8_t> hex_string_to_vector(
  const std::string &in)
try
{
  std::vector<uint8_t> out;
  boost::algorithm::unhex(in.begin(), in.end(),
    std::back_inserter(out));
  return out;
}
catch (const std::exception&)
{
  return std::vector<uint8_t>();
}
```

Listing 4

```
std::string decrypt(std::string salt, std::string iv,
  std::string encrypted, const std::string password,
  size_t iterations, const std::string& expected_mac)
{
  std::vector<uint8_t> salt_v = hex_string_to_vector(salt);
  std::vector<uint8_t> iv_v = hex_string_to_vector(iv);
  return decrypt(encrypted, password, salt_v, iv_v,
    iterations, expected_mac);
}

int main()
{
  //
  // Assuming the following message was received:
  //
  // |--------------- HMAC ---------------------|_|----------
SALT --------------||---------------- IV ----------||-------
MESSAGE ------|
  //"Fo/7rjwOjHUO0iK/REOpl4uq4L+12zA4tfc/
YnNLeTg=_be9df31d0005ebff75c68790f7730100fc588ee586cee4cc7773
27d2a010c4a1Pww/3i54DbH77FHr3+SJyg=="
  //

  std::string expected_mac =  "Fo/7rjwOjHUO0iK/
REOpl4uq4L+12zA4tfc/YnNLeTg=";
  std::string salt =
"be9df31d0005ebff75c68790f7730100";
  std::string iv =
"fc588ee586cee4cc777327d2a010c4a1";
  std::string encrypted =     "Pww/3i54DbH77FHr3+SJyg==";
  size_t iterations =         1000;
  std::string pass =          "S3cr3tP4sw";
  std::string decrypted = decrypt(salt, iv, encrypted, pass,
    iterations, expected_mac);
  std::cout << decrypted << std::endl;
}
```

Listing 4 (cont'd)

[ServCryptoLibs]: https://en.wikipedia.org/wiki/Comparison_of_cryptography_libraries

[SHA256]: https://en.wikipedia.org/wiki/SHA-256

[Stinson05]: Douglas R. Stinson (2005) *Cryptography: Theory and Practice, Third Edition* (ISBN: 1584885084)

Ex Hackina

Machine Learning and AI are popular at the moment. Teedy Deigh takes the Turing test.

Good morning.

Good morning.

What can you tell me about ML?

It's a functional programming language from the 1970s with a Hindley-Milner type system and lacking irritatingly stupid parentheses.

Sorry, perhaps I should have been clearer: what can you tell me about machine learning?

They're not very good at it. Bloody idiots, if you ask me. Always pointing out the same basic problems – same compilation errors for the same syntax errors, every time! Unable to add two and two – actually, that's about all they can do. Unable to figure out that, "No, I did not actually want to delete the whole project repo" or "No, I did not actually want to export the sweary test data into the live system", even after the fourth time... hypothetically speaking.

Artificial intelligence? Artificial, awful and amusing.

What applications do you think AI is best suited to?

Sorting out cat memes from dog memes – really, people shouldn't cross those two streams – and replacing well-understood, testable algorithms with lots of matrix multiplication over large quantities of who-knows-where-it's-from data. Great for soaking up any spare cycles on a GPU.

Why do you think developers are keen to get into AI, machine learning, etc.?

Many developers have been brought up on a staple diet of science fiction. They have also been taught that to be successful and innovative in technology they need to be disruptive. What could be more disruptive than creating Skynet, HAL or Ava? It's the perfect marriage of these two influences.

For developers who don't get to work at Facebook or Google, implementing AI functionality in their own apps probably offers the best path to disrupting the fabric of society.

Perhaps you can think of other reasons machine learning is proving popular among developers?

Of course. Developers who don't like testing their code – and who does, right? – now have the perfect excuse: because no one actually has any idea what they're expecting from a machine learning system, no one can write down their expectations, so they're off the hook!

Oh...?

I've noticed a shift from developers talking about TDD – test-driven development – to BDD – which I am predisposed to believe is bias-driven development. Because they're not entirely sure what to expect, they use *should* when describing outcomes.

Instead of implementing actual intelligence based on meaning and forms of causal and contextual reasoning – which is an AI-hard problem – an increasing number of companies have switched to creating correlation engines that reflect unintended characteristics of their input data in their output – much easier!

We're moving from GIGO being the dominant paradigm to BIBO, from garbage in, garbage out to bias in, bias out.

What can you tell me about convolutional neural networks?

I believe a convolutional neural network is a neural network with high technical debt, i.e., more spaghetti than axon. Either that, or we're talking about unnecessarily complicated solutions to simple problems. This is the complexity hill-climbing methodology implied in Anderson's law.

Anderson's law?

Named after SF author Poul Anderson: "I have yet to see any problem, however complicated, which, when you looked at it in the right way, did not become still more complicated."

It's always important to consider security in software development, and this approach clearly optimises for job security.

Speaking of accidental complexity, would you like me to send you my deep-learning solutions for the FizzBuzz, Roman numeral and bowling game katas?

Umm, no, that won't be necessary. Have you heard of Turing?

Is this a test?

Yes, and I'm afraid you haven't passed.

But I'm real! I'm Teedy! I'm a human be-

Click.

Editor's note: I received this just in time for our April edition. Somebody, as yet unknown, has interviewed our annual writer, Ms Teedy Deigh, as you can see.

Elements are reminiscent of the Voight-Kampff (VK) test from the film Blade Runner. This is used to spot replicants or androids, by focusing on emotions. The film's press-kit describes it as, "A very advanced form of lie detector that measures contractions of the iris muscle and the presence of invisible airborne particles emitted from the body. The bellows were designed for the latter function and give the machine the menacing air of a sinister insect. The VK is used primarily by Blade Runners to determine if a suspect is truly human by measuring the degree of his empathic response through carefully worded questions and statements."

The original Turing test is a way to circumvent questions about whether a machine can truly think.

Whether My Teedy Deigh has any empathy or actual intelligence remains an open question.

I can assure our readers she has not been deleted.

Or at least may be replicated in time for April 2019.

Teedy Deigh For Teedy Deigh, an existential crisis is struggling to remember the key sequence needed to get the ∃ character. She believes she is real rather than imaginary; her colleagues consider her to be complex.

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

GET MOORE
intel
Software
intel
PARALLEL STUDIO XE
£634.99
TOOLS THAT EXTEND MOORE'S LAW
CREATE FASTER CODE—FASTER
Take your results to the next level with screaming-fast code.

QBS Software Ltd is an award-winning software reseller and Intel Elite Partner

To find out more about Intel products please contact us:

020 8733 7101 | enquiries@qbssoftware.com
www.qbssoftware.com/parallelstudio

qbs SOFTWARE