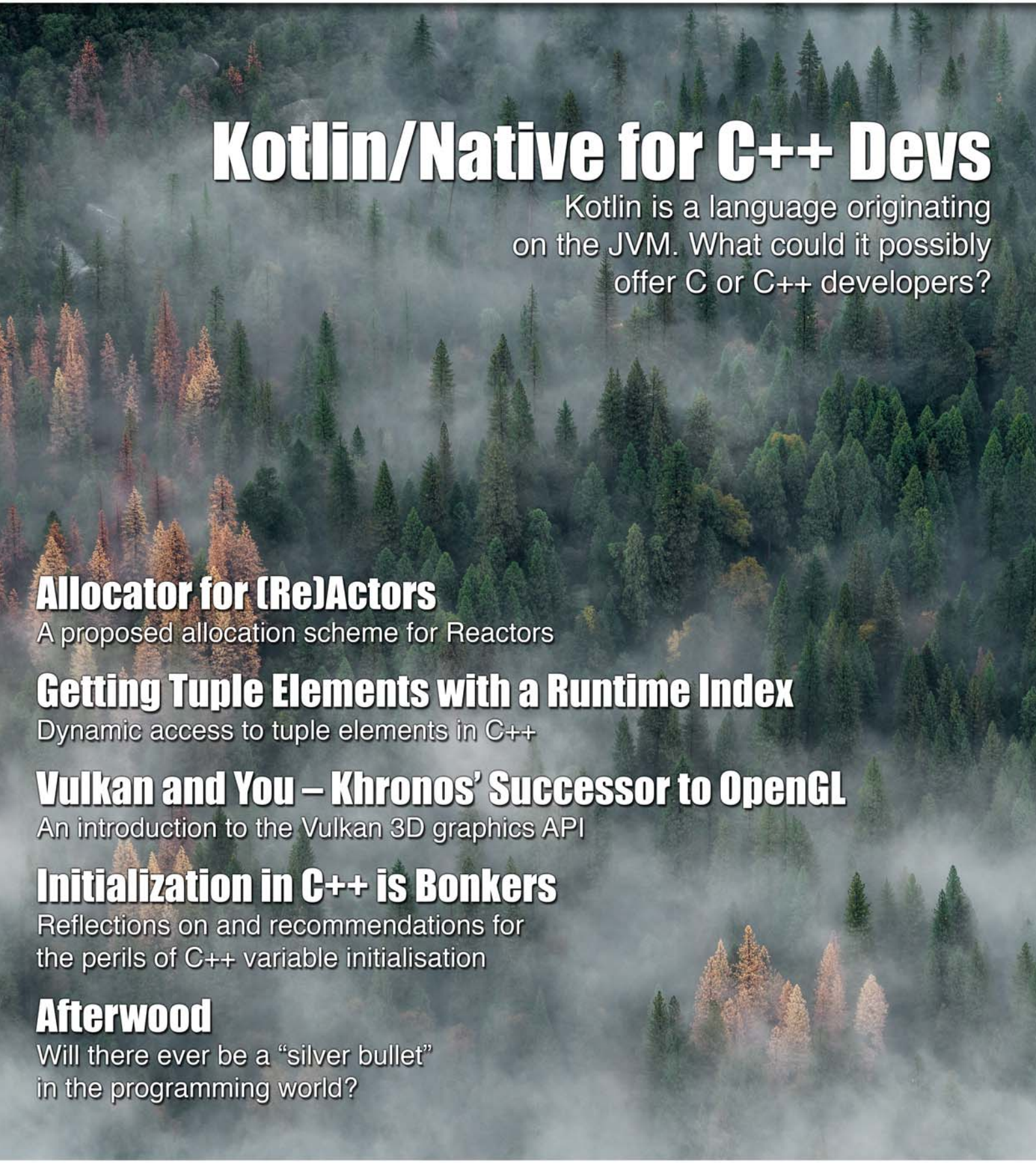
overload 139
JUNE 2017 £3



Kotlin/Native for C++ Devs

Kotlin is a language originating on the JVM. What could it possibly offer C or C++ developers?

Allocator for (Re)Actors

A proposed allocation scheme for Reactors

Getting Tuple Elements with a Runtime Index

Dynamic access to tuple elements in C++

Vulkan and You – Khronos' Successor to OpenGL

An introduction to the Vulkan 3D graphics API

Initialization in C++ is Bonkers

Reflections on and recommendations for the perils of C++ variable initialisation

Afterwood

Will there ever be a "silver bullet" in the programming world?

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 139

June 2017

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 140 should be submitted by 1st July 2017 and those for Overload 141 by 1st September 2017.

The ACCU

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

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

Overload is a publication of the ACCU

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

I am not a number

When is a number not a number? Frances Buontempo counts the ways this happens.

Distracted by my sister's photographs of her recent trip to Portmeirion [Portmeirion] recently, I am reminded of the phrase from the television programme, *The Prisoner,* which was filmed there; "I am not a number. I am a free man." In numerical computing we often see data that claims it is not a number, perhaps leaking NaNs to your front end in the process, or littering your log files. I once noticed that the Transport for London website was claiming it took **`NaN`** minutes to get from one stop to another. These things should be caught and hidden from your front end. Log files can fill up with NaNs too if you are not careful. They tend to propagate through calculations, like a virus changing all your numbers into complaints. People make mistakes when confronted with these curious beasts, a common one being trying to discover if a number is not in fact a number by trying something like comparing the number against NaN.

```
float do_some_maths()
{
  return NAN;
}
int main() {
  float quiet_nan = NAN;
  float answer = do_some_maths();
  if (quiet_nan != answer)
    std::cout << "Is a number\n";
}
```

As I hope all our readers know since a NaN is not a number it does not equal any number, furthermore it does not equal itself, so we should in fact compare **`answer != answer`**, or better yet, use standard functions like **`isnan`**, or **`double.IsNaN`**, or **`math.isnan`** or similar depending on your language.

There are many different types of NaN, at least in IEEE 754 [IEEE754], including signalling and quiet versions, and a sign, positive or negative, which may or may not mean something. I believe JavaScript just has one NaN, but the function **`isNaN`** can be applied to things like **`"123ABC"`** wherein it will tell you they are not a number and yet the empty string is a number. The Mozilla developer network [MDN] goes into glorious detail. A 'more robust' function, **`Number.isNaN()`** exists which indicates if the value is a NaN and its type is numeric, in other words it's a number that is not a number. No wonder non-technical people have such a hard time understanding what we are talking about.

If you read a file, say a csv with 0s or 1s in the columns, and add it using logstash to an index in elasticsearch[1] you might be surprised if you try to do an aggregation such as **`sum`** on a term or field, and are told the data needs to be numeric. How is a 0 or 1 not numeric? What is the world coming to if 0 or 1 is not a number? Mathematics is impossible, at least on a computer, if this is really the case. It turned out, as I expect you have already guessed, that you have to tell logstash to **`mutate`** a field if you want it to treat input as numeric rather than as a string. True story, though if I'd read the manual more carefully it would have been apparent in advance. Many newbie mistakes stem from typing a number and ending up with a string. This is unintuitive. When I type **`10`** I expect it to be **`10`**, not **`2`** let alone **`"10"`**. The majority of small children have an idea what a number is, but if you start trying to discuss strings with most adults, you are in danger of talking at cross purposes. And even after an attempt at disambiguation, the description of a small rope might not help. In one case the rope is cordage for tying things together, larger than a string in circumference, and smaller than a cable, while in the other case a rope data structure is a tree (ok, that will send us further down the rabbit hole) made of smaller strings. I am not aware of a cable data structure. Even with a clear idea of needing to be clear when a variable is a number, and being aware of the different numeric data types in your chosen language still leaves space for confusion and mistakes. To the uninitiated,

```
int x = 1,000,000;
```

looks perfectly reasonable, and yet a C++ compiler might complain about expecting an identifier and syntax error: **`"constant"`**. Say, what?! Ah, a comma is a non-numeric character so cannot be used to initialise a number without first parsing it. Yet we are used to writing numbers with separators to make them easier for us to parse. Of course, the specific character used depends on the locale, which in turn can cause problems if we write something to a file a human wants to look at and then read it back in with a computer. I recently discovered that different locales tend to use different characters as separators in, erm, 'comma' separated variable files. You could attempt to take advantage of user defined literals if you wished to express troublesome numbers in your code, or indeed use the C++14 digit separator, to say **`int x = 1'000'000;`** instead. Crawl et al provide further details in N3781 [Crawl13]. Java programmers will be laughing at this point, since they use **`_`** instead. Well, from 7 onwards. Commas make things hard to parse, it seems.

Numbers and strings differ, though it is possible to express numbers as strings, and to express some strings as numbers. We should steer clear of different types of strings and numbers otherwise we will be here forever. If you do wish to find the string equivalent of a number, we have already strayed into different locales, though just digit separators. Python draws a distinction between the **`repr()`** and **`str()`** functions. The former, giving a representation of the object, can be useful for the debugger, and

1 For non-hipsters, see https://www.elastic.co/

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.

could be passed to the evaluate function `eval` to rebuild the object. It may also contain the object's address, thereby ensuring uniqueness between different objects. In contrast, the `str()` function is designed to be slightly more friendly for humans; maybe adding some formatting to make it easier to read, or dropping extraneous information such as an object's address. Just to keep you on your toes, if you have a container of objects on which you call `str()` be aware that result will use `repr()` on the contained objects. If we return to C++, or C even, one thing which catches people out is pointers. Given `int * y` suitably initialised, printing `y` will (probably) yield something very different to printing the contents of `y`. Many beginners end up printing the address of the pointers rather than that to which it points. Why have I got this number rather than that number? I have not got the right number. Programmers more used to C# or Java may also tend to `new` objects to raw pointers and print the address by mistake. Representations, locales or idioms, if you will, change as the geography changes. Many young people complain in distress when they progress on from arithmetic in mathematics lessons to algebra, suggesting maths has no right to end up being about letters. As you gain more experience, you realise this abstraction allows you to build up general rules and discover more patterns. Furthermore, it allows programmers to write a function which takes a number using a signature like `int forward(int x);` The function takes a number, which we will refer to by a letter. Madness. We should clearly use a string like `step` instead, since single letter parameters or variable names can be a little too terse.

Let us consider integers and some basic maths. What happens when you start with `int x = 0;` and then add one, over and over. If you do this 100 times, what happens? What about a million? As we know, this rather depends or more specifically the point at which the overflow happens depends on the compiler. Is it 16-bit? 32-bit? 64-bit? Something else? Those who are paying attention will realise that had we started with an unsigned integer we would have been on safer ground, since these wrap round when they overflow, but signed integers overflowing is undefined behaviour. We can never get to +Inf by adding one over and over again. Adding one to any (unsigned) number always yields a number, but possibly a smaller one than you started with. Robert Ramey introduced a safe numerics library in a previous *Overload* [Ramey] to catch this and related issues. What happens if we initialise our `int x` with 1 and keep halving? How low can we go? Ah. Perhaps we need to make it a double or float instead. So many numbers, and so many types of numbers. What happens if we try these experiments in different languages? Can you count up to a googol in your chosen language? (Or perhaps multiply up to, since it is rather large). The general point is that the numbers you can express out of the box vary between languages. If you need larger or more precise numbers, you need to find a library, or roll your own representation.

As I am sure I have observed before, in most languages a "literal" constant, such as 5 is put inline in the code. FORTRAN puts 5 in memory which means you can change its value, for details of how see [Gorgonzola]. This is, perhaps, an extreme case of everything being an object, or at least reference. Other languages claim everything is an object. Such languages often have a `toString()` and `hashCode()` function for every object, the former of which returns a string, and the latter a number. Presumably these are both also objects, or perhaps immutable value types, or primitive data types. A talk about equality in various languages at the 2011 ACCU conference observed that Java may end up with 1,000 != 1,000, though `equals` returns true [Orr, Love]. Programming languages are odd. Programmers can be odd too, but we are all humans. We are not numbers, or resources, whatever the project plan or HR department says.

Those who use a flavour of agile may be familiar with story "points". These appear to be numeric, in fact in the sense that they provide some form of order, at least in the sense of saying this is bigger, or smaller, than that. They tend to follow a Fibonacci type sequence, rather than a linear progression to avoid a fuss over whether something is an 11 day or 12 day job. If you can choose from 1, 2, 3, 5, 8, 13 then that attempt at "precision guessing" is circumvented. Once it's got that big you could argue it's beyond hope and needs breaking down, before the team has a breakdown. A different set of cards by Lunar Logic are available just consisting of 1, TFB and NFC meaning 1, too flipping big, and no flipping chance [Lunar logic], which many readers will have come across many times before. The inexperienced will often try to convert the story points directly into days or hours, or arrange your story post-it notes into a Gantt chart. They will learn, eventually. The idea of numbers or even symbols being used to order something rather than provide a metric is important. A topology gives you relative positions; think of the London underground map, showing you which stops appear in which order along a line. In contrast, a metric gives you a "distance" (or cost or time or, erm, metric); think of the Paris metro map which shows the distance between the stations. Topologies and metrics can both use numbers, but they mean something very different. Numbers can give us an ordering, a count, a way to compare. They can also give us a description, from how many, to intensity or mass or other ideas with units; scalars versus vectors if you will. Numbers can be useful. They can be misused. They can be represented in various ways. I wonder if you can use user defined literals to cope with Roman numerals? I wish I hadn't thought of that! People, on the other hand, are neither resources nor numbers. They can be represented by numbers, say in a race, or numbers and letters, for a user id, but that is for simplicity rather than a deeper truth. I am not a number, and I still haven't written an editorial.

References

[Crawl13] 'Single quotation mark as a digit separator' Crawl, Smith, Snyder, Vandervoorde 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3781.pdf

[Gorgonzola] https://everything2.com/title/Changing+the+value+of+5+in+FORTRAN

[IEEE754] http://grouper.ieee.org/groups/754/

[Lunar Logic] https://estimation.lunarlogic.io/

[MDN] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/isNaN

[Orr, Love] https://accu.org/content/conf2011/Steve-Love-Roger-Orr-equals.pdf with more details in the *Overload* write up https://accu.org/index.php/journals/1971

[Portmeirion] http://www.portmeirion-village.com/visit/the-prisoner/

[Ramey] 'Correct Integer Operations with Minimal Runtime Penalties' Overload 137, Feb 2017 https://accu.org/index.php/journals/2344

Allocator for (Re)Actors with Optional Kinda-Safety and Relocation

How do you deal with memory for (Re)Actors? Sergey Ignatchenko proposes an allocation scheme.

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.

What is it about

As it says on the tin, this article is about allocators within the context of (Re)Actors (a.k.a. Reactors, Actors, ad hoc FSMs, Event-Driven Programs, and so on).

The main benefits we get from our C++ allocator (with (Re)Actors and proposed allocation model as a prerequisite), are the following:

- We have the option of having the best possible performance (same as that of good ol' plain C/C++)
- Without changing app-level code, we have the option of tracking access to 'dead' objects via 'dangling' pointers (causing exception rather than memory corruption in the case of such access)
- Again, without changing app-level code, we have the option of having a compactable heap. Very briefly, compacting the heap is often very important for long-running programs, as without relocation, programs are known to fall victim to so-called 'external fragmentation'. Just one very common scenario: if we allocate a million 100-byte small objects, we will use around 25,000 4K CPU pages; then if we randomly delete 900,000 of our 100-byte objects, we'll still have around 24,600 pages in use (unable to release them back to OS), just because it so happened that each of the remaining 24,600 pages has at least one non-deleted object. Such scenarios are quite common, and tend to cause quite a bit of trouble (in the example above, we're wasting about 9x more memory than we really need, *plus* we have very poor spatial locality too, which is quite likely to waste cache space and to hurt performance).
  - As a side note, many garbage-collected programming languages have been using compactable heaps for ages; I've seen this capability to compact used as an argument that garbage-collected languages are inherently better (and an argument against C++).

Let's note that while what we'll be doing allows us to achieve benefits which are *comparable* to using traditional non-C++ mark-compact garbage collectors, we're achieving those benefits in a *significantly different* manner. On the other hand, I don't want to argue whether what we're doing really qualifies as 'automated garbage collection', or if the name should be different. In the form described in this article, it is not even reference-counted garbage collection (though a similar approach can be applied to allocation models based on **`std::shared_ptr<> + std::weak_ptr<>`** – as long as we're staying within (Re)Actors).

What is important though, is to:

- Significantly reduce chances for errors/mistakes while coding.
  - Within the proposed allocation model, there are no manual **`delete`**s, which should help quite a bit in this regard.
  - In addition, the handling of 'dangling' pointers is expected to help quite a bit too (at least while debugging, but in some cases also in production).
- Allow for best-possible performance when we need it, while allowing it to be a little bit reduced (but still good enough for most production code) if we happen to need to track some bugs (or to rely on the handling of 'dangling' pointers).
- Allow for a compactable heap (again, giving *some* performance hit compared to the best-possible performance – but the performance hit should usually be mild enough to run our compactable heap in production).

Message-passing is the way to go

Before starting to speak about memory allocation, we need to define what those (Re)Actors we're about to rely on are about (and why they're so important).

For a long while, I have been a strong proponent of message-passing mechanisms over mutex-based thread sync for concurrency purposes (starting from [NoBugs10]). Fortunately, I am not alone with such a view; just as one example, the Go language's concept of "Do not communicate by sharing memory; instead, share memory by communicating" [Go2010] is pretty much the same thing.

However, only after returning from ACCU2017 – and listening to a brilliant talk [Henney17] – I realized that we're pretty much at the point of no return, and are about to reach a kinda-consensus that

> *Message-passing is THE way to implement concurrency at app-level*

(as opposed to traditional mutex-based thread sync).

The reasons for this choice are numerous – and range from "mutexes and locks are there to *prevent* concurrency" (as it was pointed out in [Henney17]), to "doing both thread sync and app-level logic at the same time tends to exceed cognitive limits of the human brain" [NoBugs15].

For the time being, it is not clear which of the message passing mechanisms will win (and whether one single mechanism will win at all) – but as I have had very good experiences with (Re)Actors (a.k.a. Actors, Reactors, ad hoc FSMs, and Event-Driven Programs), for the rest of this article I will concentrate on them.

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

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

whenever a (Re)Actor needs to communicate with another (Re)Actor ... it merely sends a message, and it is only this message which will be shared

Setting

To be a bit more specific, let's describe what I understand as (Re)Actors.

Let's use Generic Reactor as the common denominator for all our (Re)Actors. This Generic Reactor is just an abstract class, and has a pure virtual function **`react()`**:

```
class GenericReactor {
  virtual void react(const Event& ev) = 0;
}
```

Let's name any piece of code which calls GenericReactor's react() the 'Infrastructure Code'. Quite often, this call is within the so-called 'event loop':

```
std::unique_ptr<GenericReactor> r
       = reactorFactory.createReactor(...);
while(true) { //event loop
  Event ev = get_event();
    //from select(), libuv, ...
  r->react(ev);
}
```

Let's note that the **`get_event()`** function can obtain events from wherever we want – from **`select()`** (which is quite typical for servers) to libraries such as **`libuv`** (which is common for clients).

Also let's note that an event loop, such as the one above, is by far *not* the only way to call **`react()`**: I've seen implementations of Infrastructure Code ranging from one running multiple (Re)Actors within the same thread, to another one which deserialized the (Re)Actor from a DB, then called react(), and then serialized the (Re)Actor back to the DB. What's important, though, is that even if **`react()`** can be called from different threads – it MUST be called *as if* it is one single thread (='if necessary, all thread sync should be done OUTSIDE of our (Re)Actor, so **`react()`** doesn't need to bother about thread sync regardless of the Infrastructure Code in use').

Finally, let's name any specific derivative from Generic Reactor (which actually implements our **`react()`** function), a Specific Reactor:

```
class SpecificReactor : public GenericReactor {
  void react(const Event& ev) override;
};
```

Also, let's observe that whenever a (Re)Actor needs to communicate with another (Re)Actor – adhering to the 'Do not communicate by sharing memory; instead, share memory by communicating' principle – it merely sends a message, and it is only this message which will be shared between (Re)Actors.

Trivial optimization: single-threaded allocator

Armed with (Re)Actors, we can easily think of a very simple optimization for our allocation techniques. As all the processing within (Re)Actors is single-threaded, we can easily say that:

- (Re)Actor allocators can be single-threaded (i.e. without any thread sync – and avoiding relatively expensive 'compare-and-swap' operations).
  - One exception to this is those messages which the (Re)Actor sends to the others – but classes implementing those messages can easily use a different (thread-synced) allocator.
- For the purposes of this article, we'll say that each (Re)Actor will have its own private (and single-threaded) heap. While this approach can be generalized to per-thread heaps (which may be different from per-(Re)Actor heaps, in cases of multiple (Re)Actors per thread) we won't do that here.

Ok, let's write it down that our (Re)Actor allocator is single-threaded – and we'll rely on this fact for the rest of this article (and everybody who has written a multi-threaded allocator will acknowledge that writing a single-threaded one is a big relief).

However, we'll go MUCH further than this rather trivial observation.

Allocation model: owning refs, soft refs, naked refs

At this point, we need to note that in C++ (as mentioned, for example, in [Sutter11]), it is impossible to provide compacted heaps "without at least a new pointer type". Now, let's see what can be done about it.

Let's consider how we handle memory allocations within our (Re)Actor. Let's say that *within* our (Re)Actor:

- We allow for three different types of references/pointers:
  - 'owning' references/pointers, which are conceptually similar to **`std::unique_ptr<>`**. In other words, if the 'owning' reference object goes out of scope, the object referenced by it is automatically destroyed. For the time being, we can say that 'owning' references are *not* reference-counted (and therefore copying them is prohibited, though moving is perfectly fine – just as with **`std::unique_ptr<>`**).
  - 'soft' pointers/references. These are quite similar to **`std::weak_ptr<>`** (though our 'soft' references are created from 'owning' references and not from **`std:shared_ptr<>`**), and to Java WeakRef/SoftRef. However, I don't want to call them 'weak references' to avoid confusion with **`std::weak_ptr<>`** – which is pretty similar in concept, but works only in conjunction with **`std::shared_ptr<>`**, hence the name 'soft references'.
    - Most importantly – trying to dereference (in C++, call an **`operator ->()`**, **`operator *()`**, or **`operator[]`**) our 'soft' reference when the 'owning' reference is already gone is an invalid operation (leading – depending on the mode of operation – to an exception or to UB; more on different modes of operation below).
  - 'naked' pointers/references. These are just our usual C/C++ pointers.

the beauty of our memory model is that it describes WHAT we're doing, but doesn't prescribe HOW it should be implemented

- Our (Re)Actor doesn't use any non-const globals. Avoiding non-const globals is just good practice – and an especially good one in case of (Re)Actors (which are not supposed to interact beyond exchanging messages).
- Now, we're saying that *whatever forms the state of our (Re)Actor (in fact – it is all the members of our SpecificReactor) MUST NOT have any naked pointers or references (though both 'owning' and 'soft' references are perfectly fine*). This is quite easy to ensure – and is extremely important for us to be able to provide some of the capabilities which we'll discuss below.
- As for collections – we can easily say that they're exempt from the rules above (i.e. we don't care how collections are implemented – as long as they're working). In addition, memory allocated by collections may be exempt from other requirements discussed below (we'll note when it happens, in appropriate places).

With this memory allocation model in mind, I am very comfortable to say that

> *It is sufficient to represent ANY data structure, both theoretically and practically*

The theoretical part can be demonstrated by establishing a way to represent an arbitrary graph with our allocation model. This can be achieved via two steps: (a) first, we can replace all the refs in an arbitrary graph by 'soft' refs, and (b) second, there is always a set of refs which make all the nodes in the graph reachable exactly once; by replacing exactly this second set of references with our 'owning' refs, we get the original arbitrary graph represented with our 'owning refs'+'soft refs'.

As for a practical part – IMO, it is quite telling that I've seen a very practical over-a-million-LOC codebase which worked exactly like this, and it worked like a charm too.

BTW,

> *most of the findings in this article are also applicable to a more-traditional-for-C++11-folks allocation model of 'shared ptr'+'weak ptr'*

(though for single-threaded access, so atomic requirements don't apply; also, we'll still need to avoid 'naked' pointers within the state of our (Re)Actor). However, it is a bit simpler to tell the story from the point of view of 'owning' refs +'soft' refs, so for the time being we'll stick to the memory allocation model discussed above.

An all-important observation

Now, based on our memory allocation model, we're able to make an all-important

Observation 1. Whenever our program counter is within the Infrastructure Code but is outside of `react()`, there are no 'naked pointers' to (Re)Actor's heap.

This observation directly follows from a prohibition on having 'naked pointers' within (Re)Actor's state: when we're outside of `react()`, there are no 'naked pointers' (pointing to the heap of our (Re)Actor) on the stack; and as there are no non-const globals, and there are 'naked pointers' within the heap itself either – well, we're fine.

Modes of operation

Now, let's see what how we can implement these 'owning refs' and 'soft refs'. Actually, the beauty of our memory model is that it *describes* WHAT we're doing, but *doesn't prescribe* HOW it should be implemented. This leads us to several possible implementations (or 'modes of operation') for 'owning refs'/'soft refs'. Let's consider some of these modes.

'Fast' mode

In 'Fast' mode, 'owning refs/pointers' are more or less `std::unique_ptr<>`s – and 'soft refs/pointers' are implemented as simple 'naked pointers'.

With this 'fast' mode, we get the best possible speed, but we don't have any safety or reallocation goodies. Still, it might be perfectly viable for some production deployments where speed is paramount (and crashes are already kinda-ruled out by thorough testing, running new in production in 'safe' mode for a while, etc. etc.).

'kinda-Safe' mode

In a 'kinda-Safe' mode, we'll be dealing with 'dangling pointers'; the idea is to make sure that 'dangling pointers' (if there are any) don't cause memory corruption but cause an exception instead.

First of all, let's note though that because of the semantics of 'owning pointers', they cannot be 'dangling', so we need to handle only 'soft' and 'naked' pointers, and references.

'Dangling' soft references/pointers

To deal with 'dangling' soft-pointers/references, we could go the way of double-reference-counting (similar to the one done by `std::weak_ref<>` – which actually uses the ages-old concept of *tombstones*), but we can do something better (and BTW, the same technique *might* be usable to implement `std::weak_ref<>` too – though admittedly generalizing our technique to multi-threaded environment is going to be non-trivial).

Our idea will be to:

- Say that our allocator is a 'bucket allocator' or 'slab allocator'. What's important is that if there is an object at memory address X, then there cannot be an object crossing memory address X, *ever*.
  - Let's note that memory allocated by collections for their internal purposes is exempt from this requirement (!).
- Say that each allocated object has an ID – positioned right before the object itself. IDs are just incremented forever-and-ever for each new allocation (NB: 64-bit ID, being incremented 1e9 times per second, will last without wraparound for about 600 years – good enough for most of the apps out there if you ask me).

- Each of our 'owning refs' and 'soft refs', in addition to the pointer, contains an ID of the object it is supposed to point to.
- Whenever we need to access our 'owning ref' or 'soft ref' (i.e. we're calling `operator ->()` or `operator *()` to convert from our ref to naked pointer), we're reading the ID from our ref, AND reading the ID which is positioned right before the object itself – and comparing them. If there is a mismatch, we can easily raise an exception (as the only reason for such a mismatch is that the object has been deleted).
  - This approach has an inherent advantage over a tombstone-based one: as we do not need an extra indirection – this implementation is inherently more cache friendly. More specifically, we're not risking an extra read from L3 cache or, Ritchie forbid, from main RAM, and the latter can take as much as 150 CPU cycles easily. On the other hand, for our ID-reading-and-comparing, we'll be usually speaking only about the cost of 2–3 CPU cycles.

NB: of course, it IS still possible to use double-ref-counting/tombstones to implement 'kinda-Safe mode' – but at this time, I prefer an ID-based implementation as it doesn't require an extra indirection (and such indirections, potentially costing as much as 150 cycles, can hurt performance pretty badly). OTOH, if it happens that for some of the real-world projects tombstones work better, it is always still possible to implement 'kinda-Safe mode' via a traditional tombstone-based approach.

'Dangling' naked references/pointers

With naked references/pointers – well, strictly speaking, we cannot provide strict guarantees on their safety (that's why the mode is 'kinda-Safe', and not 'really-Safe'). However, quite a few measures are still possible to both detect such accesses in debugging, *and* to mitigate the impact if it happens in production:

- Most importantly, our allocation model already has a restriction on life time of 'naked' pointers, which already significantly lowers the risks of 'naked' pointers dangling around.
- In addition, we can ensure that within our (Re)Actor allocator, we do NOT really free memory of deleted objects (leaving them in a kind of 'zombie' state) – that is, until we're out of the `react()` function. This will further reduce risks of memory corruption due to a 'dangling' pointer (just because within our memory allocation model, *all* the dangling naked pointers will point to 'zombie' objects and nothing but 'zombie' objects). As for increased memory usage due to delayed reclaiming of the memory – in the vast majority of use cases, it won't be a problem because of a typical `react()` being pretty short with relatively few temporaries.
  - In debug mode, we may additionally fill deleted objects with some garbage. In addition, when out of `react()`, we can detect that the garbage within such deleted objects is still intact; for example, if we filled our deleted objects with 0xDEAD bytes, we can check that after leaving `react()` deleted objects still have the 0xDEAD pattern – and raise hell if they don't (messing with the contents of supposedly deleted objects would indicate severe problems within the last call to `react()`).
  - In production mode, we can say that our destructors leave our objects in a 'kinda-safe' state; in particular, 'kinda-safe' state may mean that further pointers (if any) are replaced with `nullptr`s (and BTW, within our memory allocation model, this may be achieved by enforcing that destructors of 'owning pointers/refs' and 'soft pointers/refs' are setting their respective pointers to `nullptr`s; implementing 'kinda-safe' state of collections is a different story, though, and will require additional efforts).
    - This can help to contain the damage if a 'dangling' pointer indeed tries to access such a 'zombie' object – at least we won't be trying to access any further memory based on garbage within the 'zombie'.

'Safe with relocation' mode

In a 'Safe with relocation' mode, in addition to dealing with 'dangling' soft refs, we'll be allowing to relocate our allocated objects. This will allow us to eliminate dreaded 'external fragmentation' – which tends to cause quite a bit of trouble for long-running systems – with lots of CPU pages having a single object in them being allocated some memory (which in turn, if we cannot possibly relocate those single objects, tends to cause *lots* of memory waste).

To implement relocation, in addition to the trickery discussed for 'Safe' mode, we'll be doing the following:

- All relocations will happen only outside of the `react()` function (i.e. when there are no 'naked' pointers to the heap, phew)
  - How exactly to relocate objects to ensure freeing pages is outside the scope of this article; here, we are concentrating only on the question of how to ensure that everything works *after* we're done relocating some of our objects
- Keep a per-(Re)Actor-heap 'relocation map' – a separate map of object IDs (the ones used to identify objects, as discussed in 'Safe' mode) into new addresses.
  - To keep the size of 'relocation map' from growing forever-and-ever, we could:
    - For each of our heap objects, keep a counter of all the 'owning' and 'soft' pointers to the object.
    - Whenever we relocate object, copy this counter to the 'relocation map'. Here, it will have the semantics of 'remaining pointers to be fixed'.
    - Whenever we update our 'owning' or 'soft' pointer as described below, decrement the 'remaining pointers to be fixed' counter (and when it becomes zero, we can safely remove the entry from our 'relocation map').
  - An alternative (or complementing) approach is to rely on 'traversing', as described below.
  - Exact implementation details of the 'relocation map' don't really matter much; as it is accessed only very infrequently, search times within it are not important (though I am *not* saying we should use linear search there).
- Whenever we detect access to a non-matching object ID (i.e. an 'owning pointer' or 'soft pointer' tries to convert to a 'naked' pointer and finds out that the object ID in heap is different from the ID they have stored), instead of raising an exception right away, we'll look into the 'relocation map' using the object ID within the pointer trying to access the object, and then:
  - If the object with such an object ID is found in the 'relocation map', we update our 'owning pointer' or 'soft pointer' to a new value and continue.
  - If the object with the ID within the pointer is not found, the object has been deleted, so we raise exception to indicate access attempt to a deleted object (just as for 'safe mode' above).
- If our relocation has led to a page being freed (and decommitted), attempts to dereference 'owning pointers' or 'soft pointers' may cause a CPU access violation. In such cases, we should catch the CPU exception, and once again look into our 'relocation map' using exactly the same logic as above (and causing either updating the current pointer, or an app-level exception).
  - To make sure that our system works as intended (and that all the pointers can still rely on an object ID always being before the object), we need to take the following steps:
    - After decommitting the page, we still need to keep address space for it reserved.
    - In addition, we need to keep track of such decommitted-but-reserved pages in a some kind of 'page map', and make sure that if we reuse the same page, we use it *only* for allocations of exactly the same 'bucket size' as before.

- While this might sound restrictive, for practical x64 systems it is usually not a big deal because (as we're decommitting the page) we'll be wasting only *address* space, and not *actual memory*. As modern x64 OSs tend to provide processes with 47-bit address space, this means that for a program which uses not more than 100G of RAM at any given time, and uses 100 different bucket sizes, in the very worst case, we'll waste at most 10000G of address space, and this is still well below that 47-bit address space we normally have.

Bingo! We've got (kinda-)safe implementation – and with the ability to compact our heap too, if we wish.

Traversing SpecificReactor state

In spite of all our efforts discussed above, in certain cases, there *might* be situations when the size of our 'page map' and especially 'relocation map' will grow too large. While I expect such situations to be *extremely* rare, it is still nice to know that there is a way to handle them.

If we say that for every object within our class `SpecificReactor`, there is a `traverse()` function (with `traverse()` at each level doing nothing but calling `traverse()` for each of child objects) then after calling `traverse()` for the whole `SpecificReactor`, we can be sure that *all* the pointers have been dereferenced, and therefore were fixed if applicable; as a result – after such a `traverse()` – our 'relocation map' is no longer necessary and can be cleaned (BTW, if we're doing `traverse()` frequently enough, we may avoid storing the reference count, which was mentioned above in the context of cleaning up the 'relocation map').

Moreover, after such a call to `SpecificReactor::traverse()`, we can be sure that there are no more pointers to decommitted pages, which means that 'page map' can be cleaned too.

On the one hand, let's note that for (Re)Actors with a large state, traversing the whole state may take a while (especially if the state is large enough to spill out of the CPU caches) – which may be undesirable for latency-critical apps. On the other hand, in such cases it is *usually* possible to implement traversing in an incremental manner (relying on the observation that any newly created objects are not a problem) – but all methods I know for such incremental traversals require us to be very careful about object moves (from a not-traversed-yet into a supposedly-already-traversed area) and about invalidating collection iterators. Still, it is *usually* possible and fairly easy to write such an incremental traversal – albeit an ad hoc one (i.e. taking the specifics of the app into account).

Further discussion planned

Actually, this is not the end of discussion about (Re)Actors and their allocators. In particular, I hope to discuss how to use such allocators to implement (Re)Actor serialization (and as mentioned in [NoBugs17], serialization of the (Re)Actor state is necessary to achieve quite a few (Re)Actor goodies, including such big things as Replay-Based Regression Testing and production post-factum debugging). ■

References

[Go2010] 'Share Memory By Communicating', The Go Blog, https://blog.golang.org/share-memory-by-communicating

[Henney17] Kevlin Henney, ACCU2017, 'Thinking Outside the Synchronisation Quadrant'

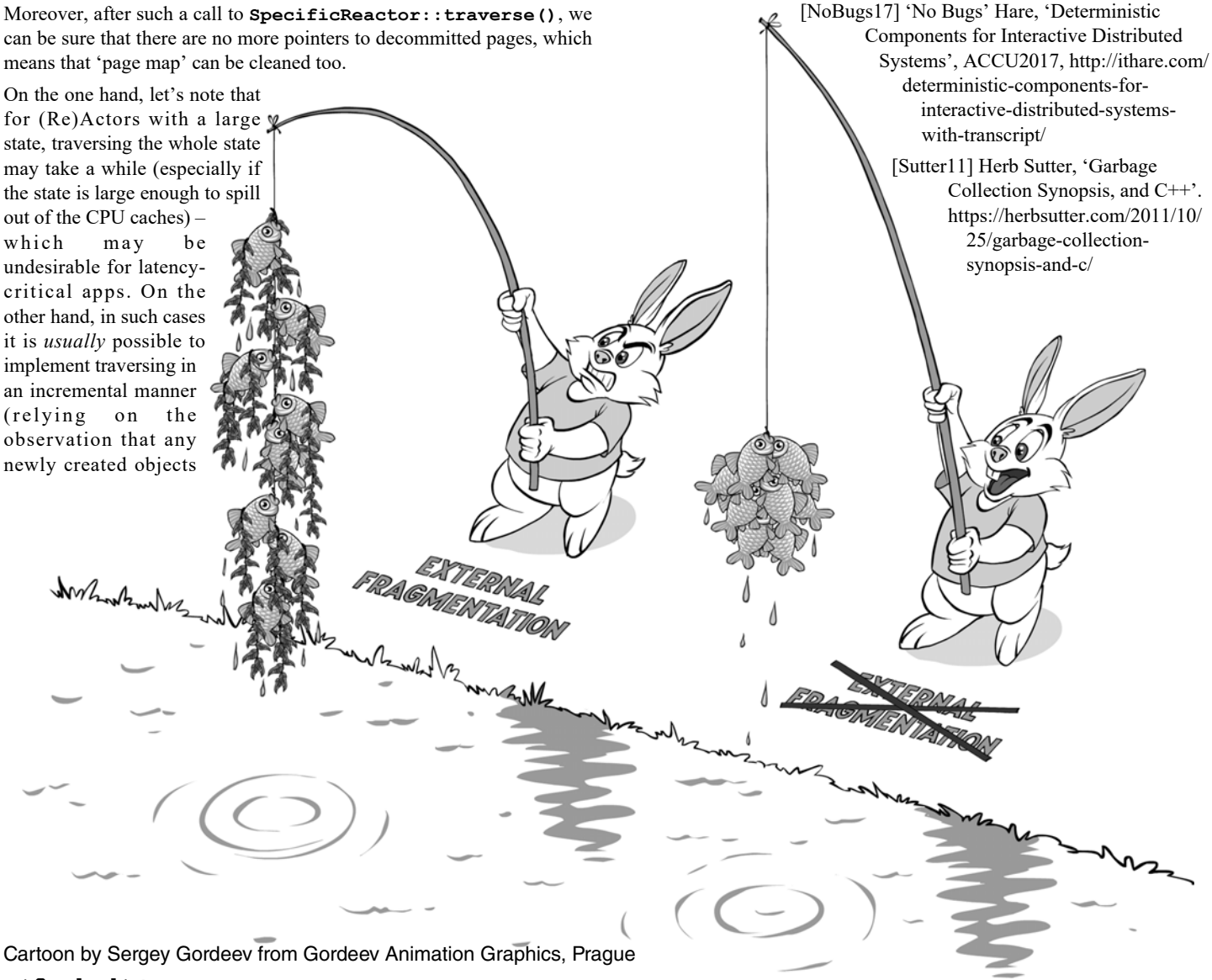
[Loganberry04] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[NoBugs10] 'No Bugs' Hare, 'Single Threading: Back to the Future?', *Overload* #97–98, June–Aug 2010

[NoBugs15] 'No Bugs' Hare, 'Multi-threading at Business-logic Level is Considered Harmful', *Overload* #128, Aug 2015

[NoBugs17] 'No Bugs' Hare, 'Deterministic Components for Interactive Distributed Systems', ACCU2017, http://ithare.com/deterministic-components-for-interactive-distributed-systems-with-transcript/

[Sutter11] Herb Sutter, 'Garbage Collection Synopsis, and C++'. https://herbsutter.com/2011/10/25/garbage-collection-synopsis-and-c/



Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague

Initialization in C++ is Bonkers

Uninitialised variables can cause problems. Simon Brand reminds us how complicated it can get.

C++ pop quiz time: what are the values of `a.a` and `b.b` on the last line in `main` of this program? (Listing 1)

```
#include <iostream>

struct foo {
    foo() = default;
    int a;
};
struct bar {
    bar();
    int b;
};
bar::bar() = default;

int main() {
    foo a{};
    bar b{};
    std::cout << a.a << ' ' << b.b;
}
```

Listing 1

The answer is that `a.a` is `0` and `b.b` is indeterminate, so reading it is undefined behaviour. Why? Because initialization in C++ is bonkers.

Default-, value-, and zero-initialization

Before we get into the details which cause this, I'll introduce the concepts of default-, value- and zero-initialization. Feel free to skip this section if you're already familiar with these (Listing 2).

```
T global;     //zero-initialization, then
              // default-initialization
void foo() {
  T i;        //default-initialization
  T j{};      //value-initialization (C++11)
  T k = T(); //value-initialization
  T l = T{}; //value-initialization (C++11)
  T m();      //function-declaration
  new T;      //default-initialization
  new T();    //value-initialization
  new T{};    //value-initialization (C++11)
}

//t is value-initialized
struct A { T t; A() : t() {} };
//t is value-initialized (C++11)
struct B { T t; B() : t{} {} };
//t is default-initialized
struct C { T t; C()       {} };
```

Listing 2

The rules for these different initialization forms are fairly complex, so I'll give a simplified outline of the C++11 rules (C++14 even changed some of them, so those value-initialization forms can be aggregate initialization). If you want to understand all the details of these forms, check out the relevant cppreference.com articles, or see the standards quotes at the bottom of the article.

- **default-initialization** – If `T` is a class, the default constructor is called; if it's an array, each element is default-initialized; otherwise, no initialization is done, resulting in indeterminate values. [cppref1]
- **value-initialization** – If `T` is a class, the object is default-initialized (after being zero-initialized if `T`'s default constructor is not user-provided/deleted); if it's an array, each element is value-initialized; otherwise, the object is zero-initialized. [cppref2]
- **zero-initialization** – Applied to static and thread-local variables before any other initialization. If `T` is scalar (arithmetic, pointer, enum), it is initialized from `0`; if it's a class type, all base classes and data members are zero-initialized; if it's an array, each element is zero-initialized. [cppref3]

Taking the simple example of `int` as `T`, `global` and all of the value-initialized variables will have the value `0`, and all other variables will have an indeterminate value. Reading these indeterminate values results in undefined behaviour.

Back to our original example

Now we have the necessary knowledge to understand what's going on in my original example. Essentially, the behaviours of `foo` and `bar` are changed by the different location of `=default` on their constructors. Again, the relevant standards passages are down at the bottom of the article if you want them, but the gist is this:

Since the constructor for `foo` is defaulted on its first declaration, it is not technically *user-provided* – I'll explain what this term means shortly, just accept this standardese for now. The constructor for `bar`, conversely, is only defaulted at its definition, so it *is* user-provided. Put another way, if you don't want your constructor to be user-provided, be sure to write `=default` when you declare it rather than define it like that elsewhere. This rule makes sense when you think about it: without having access to the definition of a constructor, a translation unit can't know if it is going to be a simple compiler-generated one, or if it's going to send a telegram to the Moon to retrieve some data and block until it gets a response.

The default constructor being user-provided has a few consequences for the class type. For example, you can't default-initialize a const-qualified object if it lacks a user-provided constructor, the notion being that if the

Simon Brand Simon is a GPGPU toolchain developer at Codeplay Software in Edinburgh. He turns into a metaprogramming fiend every full moon, when he can be found bringing compilers to their knees with template errors and writing posts for his blog at blog.tartanllama.xyz. Contact Simon at simonrbrand@gmail.com

Internalising this way of thinking about initialization is key to writing unsurprising code

object should only be set once, it better be initialised with something reasonable:

```
//ill-formed, no user-provided constructor
const int my_int;

//well-formed, has a user-provided constructor
const std::string my_string;

//ill-formed, no user-provided constructor
const foo my_foo;

//well-formed, has a user-provided constructor
const bar my_bar;
```

Additionally, in order to be trivial (and therefore POD) or an aggregate, a class must have no user-provided constructors. Don't worry if you don't know those terms, it suffices to know that whether your constructors are user-provided or not modifies some of the restrictions of what you can do with that class and how it acts.

For our first example, however, we're interested in how user-provided constructors interact with initialization rules. The language mandates that both `a` and `b` are value-initialized, but only `a` is additionally zero-initialized. Zero-initialization for `a` gives `a.a` the value `0`, whereas `b.b` is not initialized at all, giving us undefined behaviour if we attempt to read it. This is a very subtle distinction which has inadvertently changed our program from executing safely to summoning nasal demons/eating your cat/ordering pizza/your favourite undefined behaviour metaphor.

Fortunately, there's a simple solution. At the risk of repeating advice which has been given many times before, **initialize your variables**.

Seriously.

Do it.

INITIALIZE YOUR GORRAM VARIABLES.

If the designer of `foo` and `bar` decides that they should be default constructible, they should initialize their contents with some sensible values. If they decide that they should *not* be default constructible, they should delete the constructors to avoid issues. (See Listing 3.)

```
struct foo {
  foo() : a{0} {} //initialize to 0 explicitly
  int a;
};

struct bar {
  bar() = delete; //delete constructor
  //insert non-default constructor which does
  // something sensible here
  int b;
};
```

Listing 3

Internalising this way of thinking about initialization is key to writing unsurprising code. If you've profiled your code and found a bottleneck caused by unnecessary initialization, then sure, optimise it, but you best be certain that the extra performance is worth the possible headaches and money spent to keep the code safe.

If you still aren't convinced that C++ initialization rules are crazy-complex, take a minute to think of all the forms of initialization you can think of. My answers are below.

Done? How many did you come up with? In perusal of the standard, I counted eighteen different forms of initialization[1]. Here they are with a short example/description:

- default: `int i;`
- value: `int i{};`
- zero: `static int i;`
- constant: `static int i = some_constexpr_function();`
- static: zero- or constant-initialization
- dynamic: not static initialization
- unordered: dynamic initialization of class template static data members which are not explicitly specialized
- ordered: dynamic initialization of other non-local variables with static storage duration
- non-trivial: when a class or aggregate is initialized by a non-trivial constructor
- direct: `int i{42}; int j(42);`
- copy: `int i = 42;`
- copy-list: `int i = {42};`
- direct-list: `int i{42};`
- list: either copy-list or direct-list
- aggregate: `int is[3] = {0,1,2};`
- reference: `const int& i = 42; auto&& j = 42;`
- implicit: default or value
- explicit: direct, copy, or list

Don't try to memorise all of these rules; therein lies madness. Just be careful, and keep in mind that C++'s initialization rules are there to pounce on you when you least expect it. If you won't listen to me, then maybe you'll listen to the illustrious authors of the C++ Core Guidelines [cppcore], who also recommend always initializing your variables in item ES.20. And if you ever fall in to the trap of thinking C++ is a sane language, remember this:

In C++, you can give your program undefined behaviour by changing the point at which you tell the compiler to generate something it was probably going to generate for you anyway. ■

1. Feel free to debate that some of these are different flavours of initialization forms, or attributes of initialization rather than separate concepts, I don't really care, suffice to say there are a lot.

In C++, you can give your program undefined behaviour by changing the point at which you tell the compiler to generate something it was probably going to generate for you anyway

Standards quotes

All quotes from N4140 (essentially C++14).

[dcl.fct.def.default]/5:

> Explicitly-defaulted functions and implicitly-declared functions are collectively called defaulted functions, and the implementation shall provide implicit definitions for them (12.1 12.4, 12.8), which might mean defining them as deleted. **A function is user-provided if it is user-declared and not explicitly defaulted or deleted on its first declaration.** A user-provided explicitly-defaulted function (i.e., explicitly defaulted after its first declaration) is defined at the point where it is explicitly defaulted; if such a function is implicitly defined as deleted, the program is ill-formed.

[dcl.init]/6-8:

> To *zero-initialize* an object or reference of type `T` means:
>
> - if `T` is a scalar type (3.9), the object is initialized to the value obtained by converting the integer literal 0 (zero) to T
> - if `T` is a (possibly cv-qualified) non-union class type, each non-static data member and each base-class subobject is zero-initialized and padding is initialized to zero bits;
> - if `T` is a (possibly cv-qualified) union type, the object's first non-static named data member is zero-initialized and padding is initialized to zero bits;
> - if `T` is an array type, each element is zero-initialized;
> - if `T` is a reference type, no initialization is performed.
>
> To *default-initialize* an object of type `T` means:
>
> - if `T` is a (possibly cv-qualified) class type (Clause 9), the default constructor (12.1) for T is called (and the initialization is ill-formed if T has no default constructor or overload resolution (13.3) results in an ambiguity or in a function that is deleted or inaccessible from the context of the initialization);
> - if `T` is an array type, each element is default-initialized;
> - otherwise, no initialization is performed. If a program calls for the default initialization of an object of a const-qualified type `T`, `T` shall be a class type with a user-provided default constructor.
>
> To *value-initialize* an object of type T means:
>
> - if `T` is a (possibly cv-qualified) class type (Clause 9) with either no default constructor (12.1) or a default constructor that is user-provided or deleted, then the object is default-initialized;
> - if `T` is a (possibly cv-qualified) class type without a user-provided or deleted default constructor, then the object is zero-initialized and the semantic constraints for default-initialization are checked, and if `T` has a non-trivial default constructor, the object is default-initialized;
> - if `T` is an array type, then each element is value-initialized;
> - otherwise, the object is zero-initialized.

[basic.start.init]/2:

> Variables with static storage duration (3.7.1) or thread storage duration (3.7.2) shall be zero-initialized (8.5) before any other initialization takes place. […]

References

[cppcore] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Res-always

[cppref1] value-initialization http://en.cppreference.com/w/cpp/language/value_initialization

[cppref2] default-initialization http://en.cppreference.com/w/cpp/language/default_initialization

[cppref3] zero-initialization http://en.cppreference.com/w/cpp/language/zero_initialization

This article was previously published at http://blog.tartanllama.xyz/c++/2017/01/20/initialization-is-bonkers/

Vulkan and you – Khronos' successor to OpenGL

Various graphics APIs exist. Andy Thomason unravels the mysteries of Vulkan, the latest 3D Graphics API from Khronos, the custodians of OpenGL.

I love it when you get a new toy, unwrapping the box and staring for hours at the instructions while you try to put it together. Vulkan, the new graphics API from the lovely people at Khronos was a bit like that a year ago when I started to get to grips with it and like an self-assembly wardrobe, it took a lot of head scratching before it finally clicked and I was able to start making some real applications. I would not call myself an expert yet, but I may be able to explain how it works to someone who is just getting started like I was.

If you feel enthusiastic, the real reference to this is the Vulkan Spec which comes in several flavours including this one with extensions: https://www.khronos.org/registry/vulkan/specs/1.0-extensions/html/vkspec.html

Vulkan is derived from the latest OpenGL standard. Early versions of OpenGL used a fixed function pipeline and this kind of code (now obsolete) will have been familiar:

```
glBegin(GL_TRIANGLES); // Begin drawing triangles
  glVertex3f(-1, -1, 0); // Add a vertex
  glVertex3f( 0,  1, 0);
  glVertex3f( 1, -1, 0);
glEnd();
```

Later versions of OpenGL moved from fixed function pipelines to programmable shaders and the vertices moved into buffers held on the GPU and the shader parameters became uniforms: values that stayed the same for the whole object we are drawing.

```
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(Vertex),
  (void*)0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
  indexBuffer);
glDrawElements(GL_TRIANGLES, 3,
  GL_UNSIGNED_SHORT, (void*)0);
```

Vulkan's history started in 2014 when after a meeting at Valve, Khronos announced the project at Siggraph. Since then it has had contributions from Samsung, AMD and ARM to name but a few. Vulkan makes OpenGL completely stateless, more like Microsoft's DirectX, so that descriptions of objects can be made in memory and drawn in any order on multiple CPU cores if necessary. Vulkan does very little that OpenGL can't do, but it does everything in a much more modern way.

There is now the choice of a C interface (vulkan.h) or a modern C++ interface (vulkan.hpp). In my humble opinion I prefer the C++ interface, but virtually all the examples on the internet use the rather verbose C API. I have a library called Vookoo which uses the C++ API and adds a few classes to make setting up Vulkan data structures a bit easier: https://github.com/andy-thomason/Vookoo

The *raison d'etre* of Vulkan is to make as few calls as possible to the API and this is achieved by wrapping mighty data structures up into single objects. This is good for performance but can be very daunting to new users. Vookoo is designed to let beginners get used to the Vulkan API in stages, taking more responsibility as time goes by.

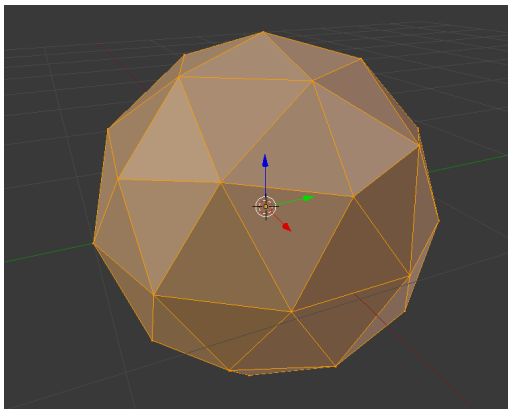
Andy Thomason Andy worked for Sony Computer Entertainment on the Playstation compilers. He now teaches game programming to aspiring developers and runs a consultancy analysing scientific data. Contact Andy at a.thomason@gold.ac.uk

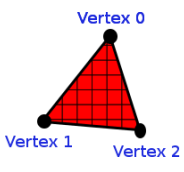
3D graphics 101

For those of you who don't work in 3D graphics, here is a helpful introduction.

The one thing that *Call of Duty* and *Angry Birds* have in common is that everything is made of triangles. That is absolutely everything, the characters, the environment, the text displaying the scores, everything. This makes it very easy to understand 3D graphics because once you can draw one triangle, you can draw everything.

To draw a triangle we need two things, a set of points in 2D or 3D space to tell us where the corners of the triangles are called 'vertices' and a bunch of numbers to tell us which of the vertices to use called the 'indices'. We have three indices per triangle we draw and can describe everything from a sprite to Lara Croft using this model. When we want to draw a 3D model on the 2D screen we use a little chunk of code that runs on the GPU called a 'vertex shader' that converts these vertices and indices to 2D triangles and calculates the lighting. After this, the GPU takes three vertices and creates a bunch of pixels. The colour of each of these is determined by a 'fragment shader' which is a function that returns red, green and blue values for each pixel.





In the very early days of 3D graphics, we worked out the positions of the vertices using graph paper, but now we have special tools like Blender to do this for us. Listing 1 is the vertex shader for our triangle; Listing 2 is the fragment shader for our triangle.

The pipeline

Graphics programming is all about the care and feeding of shaders. This is the same in all APIs be it DirectX 12, OpenGL 4.5 or Apple's

You can use any Shader language ... to generate SPIR-V ... the core specification works for all Vulkan enabled hardware

```
#version 450

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColour;
layout(location = 0) out vec3 fragColour;

void main() {
  // Copy 2D position to 3D + depth
  gl_Position = vec4(inPosition, 0.0, 1.0);
  // Copy colour to the fragment shader.
  fragColour = inColour;
}
```

Listing 1

```
#version 450

layout(location = 0) in vec3 fragColour;
layout(location = 0) out vec4 outColour;

void main() {
  // Copy interpolated colour to the screen.
  outColour = vec4(fragColour, 1);
}
```

Listing 2

proprietary Metal. The old OpenGL and DirectX APIs did a lot of their work in software, but modern graphics APIs are about getting to the hardware as quickly as possible without burning millions of cycles in drivers. Vulkan works natively on pretty much every device except for iOS and OSX but there is a proprietary adaptor from Vulkan to Metal called Molten on these.

When you execute a draw command, the index buffer selects which vertices from your model you want to assemble into a triangle and all three vertices go through the vertex stage to get moved to the right place on the screen. Say you have a model of a teapot, for example, then it consists of a few thousand (x, y, z) positions for the vertices and a few thousand indices such as (0, 1, 2). This instructs the pipeline to draw the triangles in the right place to make the teapot show up on screen in the right place.

The Vulkan pipeline is quite complex and has a few hundred parameters such as the layout of the vertices in memory and how we handle transparency. But we don't need to worry about all the detail as there are sensible defaults that just work.

SPIR-V

The shaders in Vulkan are defined by an intermediate language called SPIR-V that deserves a whole article by itself as it is both at the sharp end of Vulkan and forms the core of OpenCL, the Khronos GPU compute API. SPIR-V is a binary format with a rigid specification and that makes it easy for developers to write portable shaders. You can use any Shader language, GLSL, HLSL, CG or even C++ via LLVM to generate SPIR-V and once compiled, the core specification works for all Vulkan enabled hardware. There is a tool called 'glslangvalidator' that comes with the LunarG Vulkan SDK. This compiles GLSL shaders into SPIR-V binaries.

Shaders are fed with constants either through 'Push constants' or via memory buffers with Uniform and Vertex buffers. Push constants are good for small variables, buffers are for bigger things such as meshes or arrays of matrices for skinning characters. Shaders can also write to buffers via Storage buffers which support atomic variables. Textures are a special kind of buffer that contain images that are formatted in an opaque, optimal way such as a Hilbert curve layout to make memory accesses more local when drawing 2D images.

In Vulkan, all the textures and buffers passed to the shaders are wrapped up in a 'Descriptor Set' which is a list of handles to buffers that can be passed as a single object to the GPU, reducing the number of calls to the API.

A "hello triangle" example

This is a description of the "helloTriangle" example from Vookoo: https://github.com/andy-thomason/Vookoo/blob/master/examples/helloTriangle.cpp

You will need to install the Vulkan SDK from here: https://www.lunarg.com/vulkan-sdk/

Before we can draw a triangle, we must set up the Vulkan API. In Vookoo there is a convenient framework for the examples that will do this for you. We also create a window using the GLFW framework. Later you can explore how to do this yourself. There is a good tutorial on doing this here: https://vulkan-tutorial.com/

```
vku::Framework fw{title};
if (!fw.ok()) {
  std::cout << "Framework creation failed"
    << std::endl;
  exit(1);
}
vku::Window window{fw.instance(), fw.device(),
  fw.physicalDevice(),
  fw.graphicsQueueFamilyIndex(), glfwwindow};
```

The `vk::Device` object (`fw.device()`) is a handle to a logical device which we can use to create Vulkan objects and send commands to the GPU. `vk::PhysicalDevice` (`fw.physicalDevice()`) is the actual device and gives you information about resources available on your graphics card or phone. Each logical device supports several queues (`fw.graphicsQueueFamilyIndex()`) to send commands to the GPU. Some queues are for graphics, some for transfer etc.

Next up we need to set up the shaders.

```
vku::PipelineMaker pm{(uint32_t)width,
  (uint32_t)height};
pm.shader(vk::ShaderStageFlagBits::eVertex,
  vert_);
pm.shader(vk::ShaderStageFlagBits::eFragment,
  frag_);
pm.vertexBinding(0, (uint32_t)sizeof(Vertex));
pm.vertexAttribute(
  0, 0, vk::Format::eR32G32Sfloat,
  (uint32_t)offsetof(Vertex, pos));
pm.vertexAttribute(
  1, 0, vk::Format::eR32G32B32Sfloat,
  (uint32_t)offsetof(Vertex, colour));

auto renderPass = window.renderPass();
auto &cache = fw.pipelineCache();
auto pipeline = pm.createUnique(
  device, cache, *pipelineLayout_, renderPass);
```

Listing 3

```
vku::ShaderModule vert_{device,
  BINARY_DIR "helloTriangle.vert.spv"};
vku::ShaderModule frag_{device,
  BINARY_DIR "helloTriangle.frag.spv"};
```

These we load from binary files compiled by glslangvalidator.

```
vku::PipelineLayoutMaker plm{};
auto pipelineLayout_ = plm.createUnique(device);
```

The pipeline layout is a description of the descriptor sets used to pass buffers to shaders. In this case, we don't use one as we only pass vertices to the vertex shader.

Next we define our vertex format and make the three corners of our triangle.

```
struct Vertex { glm::vec2 pos; glm::vec3 colour;
};
const std::vector<Vertex> vertices = {
  {{0.0f, -0.5f}, {1.0f, 0.0f, 0.0f}},
  {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
  {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
};
vku::VertexBuffer buffer(fw.device(),
  fw.memprops(), vertices);
```

Now we have a triangle, we build our pipeline. The Vulkan data structure for doing this is a little verbose, so Vookoo has another helper object to make this easy (see Listing 3).

The pipeline cache object holds the binary information that gets sent to the GPU and can be saved to speed up the process of building pipelines in the future. The `renderPass` object holds information about the frame buffer we are drawing to, in this case a set of special images which will be copied to the window. It describes how we clear the frame buffer, which images we are rendering to and what to do with the frame buffer after we are done drawing.

Instead of sending commands directly to the GPU, Vulkan records commands in command buffers which are then put into queues for later asynchronous execution on the GPU. Listing 4 is a code example for setting up a command buffer to draw a single triangle using the C++ interface. We have omitted quite a bit of code for clarity.

Because Vulkan draws asynchronously, we usually use at least three almost identical command buffers to draw up to three frames in advance. Alternatively, we can allocate command buffers as we need them from a pool.

Finally we can just submit our command buffer to a queue and wait for the GPU to draw our triangle. This is done in the framework by the

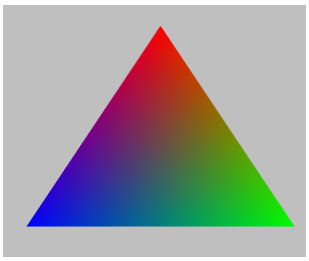
```
vk::CommandBuffer cb = ...;
vk::CommandBufferBeginInfo bi{};
cb.begin(bi);
cb.beginRenderPass(rpbi,
  vk::SubpassContents::eInline);

cb.bindPipeline(vk::PipelineBindPoint::eGraphics,
  *pipeline);
cb.bindVertexBuffers(0, buffer.buffer(),
  vk::DeviceSize(0));
cb.draw(3, 1, 0, 0);
cb.endRenderPass();
cb.end();
```

Listing 4

`window.draw()` call which also handles the synchronisation you need to do to prevent clashes on the GPU.

```
while (!glfwWindowShouldClose(glfwwindow)) {
  glfwPollEvents();
  window.draw(fw.device(), fw.graphicsQueue());
  std::this_thread::sleep_for(
    std::chrono::milliseconds(16));
}
```

If it works for you, you will be rewarded with this magnificent triangle:



The whole example is only 91 lines long, but the `vku::framework` and `vku::window` objects hide a lot of complexity. Like DirectX, Vulkan is challenging to set up at get started with but the reward is some very high performance and little drain on the CPU.

Debugging

Vulkan development would be almost impossible with the debugging layers that come with the SDK. If Vulkan encounters an error internally, it will likely crash or worse still just display nothing on the screen.

Vulkan has a system of layers that make development a lot more pleasant. You can add verification layers to detect errors in your setup or just warn you. Once your code runs without warnings and errors, you can let it loose and it will execute much faster. This is typical of the games industry where intense testing is done before releasing a title so that we can shave a few cycles off the frame time. These days with VR headsets, we want to be running at 90 frames per second without hiccups and so don't have time for niceties like exceptions and runtime error checking. That means that your game must do all the physics, AI, networking, gameplay and rendering in 11ms or about 33 million cycles.

And so...

Vulkan is definitely fun once you have got past the pain of setting up the API. I think that we can teach it to students instead of OpenGL now despite the additional complexity. We hope that the remaining holdouts will support Vulkan on consoles and devices so that we can all code to a common standard.

Interestingly, Vulkan works especially well with Mobile devices as the `renderPass` structure is well suited to tiled GPUs such as we find on phones, tablets and increasingly TVs and VR headsets and so has a brilliant future there.

Live long and prosper, as they say… ■

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

Kotlin for C++ Developers

What could a language the originated on the JVM possibly offer C or C++ devs? Hadi Hariri tells us.

Kotlin is a programming language that was initiated by JetBrains, makers of developer tools such as IntelliJ IDEA, CLion, ReSharper et al. It started in 2010 as a need for JetBrains to use a language that was less verbose than Java, had 'modern' constructs, good tooling, and led to more maintainable code in the long run. After looking at the alternatives at the time, a decision was made to write their own.

Fast-forward to February 2017 and Kotlin released version 1.1 which targets not only the JVM but JavaScript. With an impressive growth in adoption since the release of the first version in February 2016, Kotlin is now being used by companies large and small, including Netflix, Gradle, American Express, Expedia, Pinterest, Trello, BBC, just to name a few.

In March 2017, JetBrains, we, announced the first Technology Preview of Kotlin/Native, which is Kotlin without any kind of virtual machine, targeting via LLVM multiple platforms, including

- Mac OS X 10.10 and later (x86-64)
- x86-64 Ubuntu Linux (14.04, 16.04 and later), other Linux flavours may work as well
- Apple iOS (arm64), cross-compiled on macOS host
- Raspberry Pi, cross-compiled on Linux host

Kotlin is a language that is designed to be used at industrial scale, targeting any type of application development, be it backend, desktop, front-end on the web as well as mobile. And of course, now with native, other possibilities come into play such as embedded and Internet of Things.

As a C or C++ developer though, what does Kotlin, and in particular Kotlin/Native, have to offer? To answer that, let's take a brief look at some of the key features of the language.

Above all, pragmatism

The main principle behind Kotlin is pragmatism. While it is unfortunate that at times that word is used to justify shortcomings, in the case of Kotlin the purpose is to look at common situations we encounter as software developers when using different languages and solving problems, and try and address these.

To give some examples – in the business world we often use the concept of data transfer objects, a class that holds some properties but really doesn't have much when it comes to behaviour. It does, however, require certain things such as the ability to represent the data as a string, to copy from one instance to another, or to compare two instances based on properties. In Kotlin, all this can be done in a single expressive line of code:

```
data class Customer(val name: String,
val email: String, val country: String)
```

This line provides a class with three properties, that are read-only (**`val`** indicates read-only, **`var`** indicates writable), namely: **`name`**, **`email`** and **`country`**. But in addition it provides a function **`toString`**, **`equals`**, **`copy`**, and **`hashCode`**.

Conciseness appears in other ways, following the principle of not repeating oneself, Kotlin doesn't require explicit type declaration or conversion when the compiler can easily infer things. For instance, variables can be declared and initialised without explicitly declaring the type

```
val myString = "Something"
```

as opposed to the longer format

```
val myString : String = "Something"
```

When performing type conversions, there's no need to verify and cast explicitly:

```
If (myObject is Customer) {
                myObject.makeActive()
```

auto-casting takes place, as opposed to having to explicitly convert the type

```
If (myObject is Customer) {
                (Customer)myObject.makeActive()
```

Certain patterns, such as the SINGLETON pattern become extremely easy to implement in Kotlin, as it has the concept of an object (a single instance, not requiring instantiation from any class)

```
object Global {
  val version = "0.1"
  fun log(message: String) {
    println("LOG: $message")
  }
}

fun usingSingletons() {
  Global.log("A Message is sent")
}
```

Of course Kotlin isn't only about conciseness, but also provides a series of characteristics that allow for writing nice statically-typed DSLs. Being a language that treats functions as first class citizens, it allows for higher-order functions and lambda expressions

```
val countryCapital = listOf("Madrid" to "Spain",
 "London" to "UK", "Berlin" to "Germany",
 "Washington DC" to "USA")
val capitals = countryCapital.map {
  it.first
}.sorted()
```

The combination of extension functions, which allow us to extend an existing type with new functionality without having to inherit from it,

Hadi Hariri A developer and creator of many things OSS, his passions includes Web Development and Software Architecture. Has authored a couple of books, a few courses and has been speaking at industry events for nearly 15 years. Currently at JetBrains leading the Developer Advocacy team. Spends as much time as he can writing code. Contact him at hadi@jetbrains.com

Kotlin is completely open when it comes to tooling. You can use anything, be it the command line compiler, which is simple, up to a fully fledged IDE such as IntelliJ IDEA

lambda expressions, and a series of other features and conventions allow us to create DSLs like the following:

```
build {
  make {
    source = "*.kt"
    target  = "/tmp"
 }
}
```

where each of the elements are actual static identifiers.

Leveraging the platform

A language alone in isolation often doesn't provide the productivity someone needs. The focus on pragmatism in Kotlin also surfaces when we're speaking about interoperability with platforms and leveraging existing libraries, frameworks and in general the ecosystem present. This is demonstrated on the JVM with interoperability with Java and JVM libraries, in JavaScript with interop with package managers, JavaScript standards and of course when it comes to native, Kotlin also provides interop with C.

The code in Listing 1 demonstrates how we can interact with C libraries, in this case provide socket functionality.

Next steps

If you want to learn more about Kotlin, the best place to get started is with the Kotlin Koans, which are a series of exercises that can be performed online [https://try.kotlinlang.org] or offline [https://kotlinlang.org/docs/tutorials/koans.html]. Kotlin is completely open when it comes to tooling. You can use anything, be it the command line compiler, which is simple, up to a fully fledged IDE such as IntelliJ IDEA (works in both the free OSS Community Edition as well as the commercial Ultimate one). In addition to IntelliJ IDEA, at JetBrains we also provide support for Android Studio, Eclipse and Netbeans.

Summary

It is close to impossible to cover extensively any language in such a short amount of space. Independently of this fact, however, I've personally found that with Kotlin it is not about specific features that make it stand out, but how all these different things fit in together to provide a better experience when writing and reading code. And it is this experience that can now be shared across multiple platforms with different developers of different backgrounds. As a C or C++ developer, what Kotlin can provide is a higher-level language abstraction, making code often easier to write, and more importantly understand, while at the same time not losing the power to interact with low-level constructs when needed.

Kotlin/Native still has a long way to go, but this first technology preview itself is a big milestone. The focus for the Kotlin/Native team is to continue to build on what is currently available, improving tooling, providing support for more platforms and in general providing a pleasurable development experience to all. ■

```
import kotlinx.cinterop.*
import sockets.*

fun main(args: Array<String>) {
  if (args.size < 1) {
    println("Usage: ./echo_server <port>")
    return
  }

  val port = atoi(args[0]).toShort()

  memScoped {
    val bufferLength = 100L
    val buffer = allocArray<ByteVar>(bufferLength)
    val serverAddr = alloc<sockaddr_in>()
    val listenFd = socket(AF_INET, SOCK_STREAM, 0)
        .ensureUnixCallResult { it >= 0 }

    with(serverAddr) {
      memset(this.ptr, 0, sockaddr_in.size)
      sin_family = AF_INET.narrow()
      sin_addr.s_addr = htons(0).toInt()
      sin_port = htons(port)
    }

    bind(listenFd, serverAddr.ptr.reinterpret(),
      sockaddr_in.size.toInt())
      .ensureUnixCallResult { it == 0 }

    listen(listenFd, 10)
      .ensureUnixCallResult { it == 0 }

    val commFd = accept(listenFd, null, null)
      .ensureUnixCallResult { it >= 0 }

    while (true) {
      val length = read(commFd, buffer,
        bufferLength)
        .ensureUnixCallResult { it >= 0 }
      if (length == 0L) {
        break
      }

      write(commFd, buffer, length)
         .ensureUnixCallResult { it >= 0
    }
  }
}
```

Listing 1

Getting Tuple Elements with a Runtime Index

Accessing a tuple with a runtime index is a challenge. Anthony Williams shows us his approach.

`std::tuple` is great. It provides a nice, generic way of holding a fixed-size set of data items of whatever types you need. However, sometimes it has limitations that mean it doesn't quite work as you'd like. One of these is accessing an item based on a *runtime* index.

std::get needs a compile-time index

The way to get the nth item in a tuple is to use `std::get`: `std::get<n>(my_tuple)`. This works nicely, as long as `n` is a compile-time constant. If you've got a value that is calculated at runtime, this doesn't work: you can't use a value that isn't known until runtime as a template parameter.

```
std::tuple<int,int,int> my_tuple=...;
size_t index;
std::cin>>index;
int val=std::get<index>(my_tuple);//won't compile
```

So, what can we do? We need a new function, which I'll call `runtime_get`, to retrieve the `n`th value, where `n` is a runtime value.

```
template<typename Tuple>
... runtime_get(Tuple&& t,size_t index){
  ...
}
```

The question is: how do we implement it?

Fixed return type

The return type is easy: our function must have a single return type for any given `Tuple`. That means that all the elements in the tuple must have the same type, so we can just use the type of the first element. `std::tuple_element` will tell us this, though we must first adjust our template parameter so it's not a reference.

```
template<typename Tuple>
typename std::tuple_element<
 0, typename std::remove_reference<Tuple>
 ::type>::type&
runtime_get(Tuple&& t,size_t index){
  ...
}
```

Note: C++17 includes `std::variant`, so you might think we could use that to hold the return type, but that wouldn't actually help us: to get the value from a variant, you need to call `std::get<n>(v)`, which requires `n` to be a constant (again)!

OK, so the return type is just a reference to the type of the first element. How do we get the element?

Anthony Williams Anthony is the author of *C++ Concurrency in Action*. As well as working on multi-threading libraries, he develops custom software for clients, and does training and consultancy. Despite frequent forays into other languages, he keeps returning to C++. He is a keen practitioner of TDD, and likes solving tricky problems. Contact him at anthony@justsoftwaresolutions.co.uk

Retrieving the nth element

We can't do a straightforward `switch`, because that requires knowing all the cases in advance, and we want this to work for any size of tuple.

One way would be to have a recursive function that checked the runtime index against a compile-time index, and then called the function with the next compile-time index if they were different, but that would mean that the access time would depend on the index, and potentially end up with a deeply nested function call if we wanted the last element in a large tuple.

One thing we *can* do is use the index value as an array index. If we have an array of functions, each of which returns the corresponding element from the tuple, then we can call the appropriate function to return the relevant index.

The function we need is of course `std::get`; it's just a matter of getting the function signature right. Our overload of `std::get` has the following signature for `const` and non-`const` tuples:

```
template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>&
get(tuple<Types...>&) noexcept;
template <size_t I, class... Types>
constexpr const tuple_element_t<I,
  tuple<Types...>>&
get(const tuple<Types...>&) noexcept;
```

so, we can capture the relevant instantiation of `std::get` for a given tuple type `Tuple` in a function pointer declared as:

```
using return_type=typename
  std::tuple_element<0,Tuple>::type&;
using get_func_ptr=return_type(*)(Tuple&)
  noexcept;
```

The signature is the same, regardless of the index, because we made the decision that we're only going to support tuples where all the elements are the same.

This makes it easy to build a function table: use a variadic pack expansion to supply a different index for each array element, and fill in `std::get<N>` for each entry (see Listing 1).

We need the separate redeclaration of the table to satisfy a pre-C++17 compiler; with C++17 inline variables it is no longer needed.

Our final function is then just a simple wrapper around a table lookup (see Lising 2).

It's `constexpr` safe, though in a `constexpr` context you could probably just use `std::get` directly anyway.

So, there you have it: a constant-time function for retrieving the `n`th element of a tuple where all the elements have the same type.

Final code

Listing 3 is the final code for a constant-time function to retrieve an item from a tuple based on a runtime index. ■

```
template<
  typename Tuple,
  typename Indices=std::make_index_sequence<std::tuple_size<Tuple>::value>>
struct runtime_get_func_table;

template<typename Tuple,size_t ... Indices>
struct runtime_get_func_table<Tuple,std::index_sequence<Indices...>>{
    using return_type=typename std::tuple_element<0,Tuple>::type&;
    using get_func_ptr=return_type (*)(Tuple&) noexcept;
    static constexpr get_func_ptr table[std::tuple_size<Tuple>::value]={
        &std::get<Indices>...
    };
};

template<typename Tuple,size_t ... Indices>
constexpr typename
runtime_get_func_table<Tuple,std::index_sequence<Indices...>>::get_func_ptr
runtime_get_func_table<Tuple,std::index_sequence<Indices...>>::table[
  std::tuple_size<Tuple>::value];
```

Listing 1

```
template<typename Tuple>
constexpr
typename std::tuple_element<0,typename std::remove_reference<Tuple>::type>::type&
runtime_get(Tuple&& t,size_t index){
    using tuple_type=typename std::remove_reference<Tuple>::type;
    if(index>=std::tuple_size<tuple_type>::value)
        throw std::runtime_error("Out of range");
    return runtime_get_func_table<tuple_type>::table[index](t);
}
```

Listing 2

```
#include <tuple>
#include <utility>
#include <type_traits>
#include <stdexcept>

template<
  typename Tuple,
  typename Indices=std::make_index_sequence<std::tuple_size<Tuple>::value>>
struct runtime_get_func_table;

template<typename Tuple,size_t ... Indices>
struct runtime_get_func_table<Tuple,std::index_sequence<Indices...>>{
    using return_type=typename std::tuple_element<0,Tuple>::type&;
    using get_func_ptr=return_type (*)(Tuple&) noexcept;
    static constexpr get_func_ptr table[std::tuple_size<Tuple>::value]={
        &std::get<Indices>...
    };
};

template<typename Tuple,size_t ... Indices>
constexpr typename
runtime_get_func_table<Tuple,std::index_sequence<Indices...>>::get_func_ptr
runtime_get_func_table<Tuple,std::index_sequence<Indices...>>::table[std::tuple_size<Tuple>::value];

template<typename Tuple>
constexpr
typename std::tuple_element<0,typename std::remove_reference<Tuple>::type>::type&
runtime_get(Tuple&& t,size_t index){
    using tuple_type=typename std::remove_reference<Tuple>::type;
    if(index>=std::tuple_size<tuple_type>::value)
        throw std::runtime_error("Out of range");
    return runtime_get_func_table<tuple_type>::table[index](t);
}
```

Listing 3

Afterwood

What makes programming fun? Chris Oldwood ponders what floats his boat.

Back in 1986, Fred Brooks published the seminal essay *No Silver Bullet: Essence and Accident in Software Engineering* in which he introduced us to the idea that there are two different kinds of complexity which software engineers need to deal with. The first is 'essential complexity', which is complexity that is inherent in the problem being solved. Before we can solve the problem, we humans need to understand the problem domain and model it before we can think about how we're going to represent it in the solution domain. In the essay, Brooks argues that there is little to really help us quickly get into and understand the true essence of the problems we try to solve through software. Yes, there have been some advances but nothing stellar.

The second kind of complexity which Brooks described, he coined 'accidental complexity'. This exists in the solution domain and is the complexity which exists as a by-product of the processes, tools and technologies we use to solve our customer's problems. This kind of complexity could be perceived as being a problem of our own making, as the programming languages and tools we use are imperfect and therefore imprecise. These are the silver bullets which the essay is referring to.

Unfortunately, the choice of the term 'accidental' became a poor one as it was often interpreted incorrectly. In his follow-up essay almost 10 years later, *No Silver Bullet Refired*, he describes how it was often treated in the sense of 'misfortune' rather than its alternative meaning of 'incidental'. As such, many rebuttals targeted the apparent incompetence of programmers instead of the imperfections in the tooling, which missed the point of the original essay.

In theory, as we make technological advances we should be reducing the degree of accidental complexity – the ability to represent our solution for the problem in computerised form – and instead spend more time focusing on tackling the essential complexity. The growth of the Agile movement which puts an emphasis on trying to ensure we 'build the right thing' before wasting time 'building the wrong thing, in the right way' has also meant that the essential complexities are starting to get more of a look-in before we get bogged down in the coding. This is probably not surprising given that the latter part of *No Silver Bullet* lists the use of 'incremental development' and the metaphor to 'grow, not build' software as probably the most promising approach to tackling the problem.

No Silver Bullet was added to the Anniversary Edition of his book *The Mythical Man-Month: Essays on Software Engineering*, which was published 20 years later in 1995. The first essay in that book, and one I'm personally very fond of, is 'The Tar Pit'. It's one of the shorter ones, weighing in at just 6 pages, and provides a brief introduction to the rest of book by exploring what systems programming is all about. Along the way he looks at both the 'joys' and the 'woes' of the craft and suggests that for most people 'the joys far outweigh the woes', which I reckon most programmers would find little trouble agreeing with, even if on certain days it feels like the latter might be true.

And so with the lay of the land firmly established I feel comfortable in making my confession – it's the accidental complexity that I find exciting. That's right, what floats my particular boat is solving the problems which shouldn't really exist but do because of those imperfections in the tooling we use. The Internet is awash with advice on how we should get better at understanding our customer's needs and solve their problems more efficiently but quite frankly that's just not as interesting to me as trying to dig yourself out of a technical hole which either you (or your predecessor) got you into in the first place.

That doesn't mean I actively go out of my way to be obtuse or pick technologies that are unsuitable purely for the purposes of self-gratification – on the contrary I still want to be professional and do the best job I can – it's just I personally often find more pleasure solving the problems in the solution domain than solving the actual customer's problem.

One example of this would be the war stories that programmers like to share over a pint or two in the bar at a conference. The ones I'm most fond of hearing or reading about (and sharing) are those that involve one person's struggle against the technology. There are so many wonderful tales about how people have managed to bend, twist and generally contort one tool to make it do something it was never intended to. In *An Introduction to General Systems Thinking*, Gerry Weinberg suggests that you don't really know a tool until you've abused it at least three times, so perhaps there is a rite of passage where we have to embrace the gnarly problems to truly grasp the very nature of accidental complexity so that we can try to elude it in the future. As an aside, that book was originally published at the same time as *The Mythical Man Month* (and 10 years before *No Silver Bullet*).

My current best efforts to explain these pangs of guilt come from Vivek Singh (@petmongrels) who recently tweeted "Programmer Stack Envy – A belief that work at a lower level in the stack is intellectually more challenging than at one's own level". I know that when I leave the ACCU Conference every year, I am in awe of many of the people attending and the things they've accomplished, especially as the C++ language grows ever more complex with each new standard. But I wonder how many of those people are also suffering from a form of Stockholm syndrome and are in it largely because it *is* complex – an enigma to be solved in its own right.

Secretly, I hope there will never be a silver bullet because when that day comes I fear it's the day the fun goes out of programming. ■

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