overload 159
OCTOBER 2020 £4.50

Concurrency Design Patterns

Orchestrating concurrent tasks using mutexes is seldom efficient. We investigate design patterns that help unlock concurrent performance.

C++ Modules: A Brief Tour

A tourist's guide to C++20's long awaited module system

Kafka Acks Explained

Visualizing Kafka's most misunderstood configuration setting

The Edge of C++

Every technology has a boundary; we look at the "outer limits" of C++

poly::vector - A Vector for Polymorphic Objects

An efficient C++ container of polymorphic objects, based on STL principles

A magazine of ACCU
ISSN: 1354-3172

JET BRAINS

A Power Language Needs Power Tools

Smart editor with full language support
Support for C++03/C++11, Boost and libc++, C++ templates and macros.

Reliable refactorings
Rename, Extract Function / Constant / Variable, Change Signature, & more

Code generation and navigation
Generate menu, Find context usages, Go to Symbol, and more

Profound code analysis
On-the-fly analysis with Quick-fixes & dozens of smart checks

GET A C++ DEVELOPMENT TOOL THAT YOU DESERVE

ReSharper C++
Visual Studio Extension for C++ developers

AppCode
IDE for iOS and OS X development

CLion
Cross-platform IDE for C and C++ developers

Start a free 30-day trial
jb.gg/cpp-accu

Find out more at **www.qbssoftware.com**

QBS SOFTWARE

OVERLOAD 159

October 2020

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

Tor Arve Stangeland
tor.arve.stangeland@gmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 160 should be submitted by 1st November 2020 and those for Overload 161 by 1st January 2021.

The ACCU

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

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

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

Copyrights and Trade Marks

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

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

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

Virtual/Reality

Do we know what reality is? Frances Buontempo is no longer sure and now wonders if she's a fictional character.

Since becoming *Overload* editor, I have failed to write an editorial. In my defence, I have managed to fill the first two pages with musings, somewhat like a meta- or virtual editorial. Having attended a live stream of various noisy bands instead of making our annual pilgrimage to a metal festival in a field in England during August, I have again failed to find time to write an editorial. The online festival was tremendous; I discovered a couple of new bands and interacted with others across the world on social media while they played. Nonetheless, this is not the same as being in a circle pit or somewhere near the front bouncing off other audience members. Live music is a real visceral experience that is impossible to capture virtually. I tried to imagine a haptic feedback mosh-pit suit, but didn't get very far. I'll need to read some decent SciFi to plug the gaps in my imagination. Fictional accounts of possibilities frequently pave the way for changes in technology, or more broadly society in general. Fictional, or virtual, imaginings cause a shift in the fabric of reality. *Star Trek* has arguably brought about mobile phones, tablets and automatic doors. I'm hanging on for the replicators and transporters, as I have said many times before. I am given to believe that the film *Predator* [IMDb-1] caused someone in the military to ask if the mimetic camouflage suit were possible in reality, securing funding to research this. 'Now the nightmare vision of an invisible murderer from space could come true on Earth, thanks to University of Bristol scientists.' [Waugh15]. Cheerful stuff.

Over time, many people have bashed 'virtual' interactions. I have been asked, "Have you actually talked to them?" when I say I've been discussing a technical issue. Some people can't understand how to communicate by writing only, and believe a 'proper' conversation is always better. This general statement misses many nuances, and the best way to communicate almost always depends. For example, a colleague did 'phone' me, or make a Slack call rather, to talk me through running a script. As with many step-up scripts, it asked probing question, like 'Wipe all this out and replace it? [y/N]'. The technical among you will realise pressing enter will select the upper case option 'N', which is a shorthand for 'No'. As you can imagine, the actual words on the call went like:

Fran: No?
Him: Yes.
Fran: What? Yes.
Him: No, No.
Fran: Tell you what, I'll just accept the defaults and call you back.

Whoever insists that phone calls are better than typing or scripts inhabits a different reality to a large amount of my life. Did I tell you about the guy who tried to read a barcode down the phone to a customer once? "Thick, thick, thin…" I kid you not. Talking to people is not always the best way to communicate. Furthermore, I wonder if a Slack call is even a real telephone call?

A telephone is a sound at a distance. Anything starting 'tele' captures the idea of something happening at a distance, so I guess a Google Hangout, Zoom, or Slack call are like phone calls, but give the optional extra of having your web-cams on so you can see each other. This, of course, can put stress on the bandwidth, since you are uploading and downloading pictures and sounds, rather than just listening to each other. Very distracting. I acknowledge some people like to see each other and wave. Having a virtual beer with the camera on is great. You can discuss what you are drinking. Or, for the alcohol-free, sharing a remote cup of tea and cake means you can show off the cake and discuss recipes. All good. Something that works well for one situation may not work well across the board. As all consultants will tell you, there is no One True Way to approach things. It always depends.

Talking over a phone or video call is no less talking than a face to face conversation. Certainly, there are differences. You don't have to spend time and money travelling to be in the same place. You can't shake hands or hug remotely. You can still talk, and listen. Sharing barcodes or running a script might be better done without talking, as discussed. Writing things down can force you to be precise and unambiguous with language. It also provides a paper trail, which is useful in a variety of circumstances. It may not count as a **real** conversation, but who cares or even knows what's real. Life is complex.

Code can get complex too. Object oriented programming using the idea of dynamically dispatched functions, flagged up as `virtual` to vary behaviour at runtime. In *The Design and Evolution of C++* [Stroustrup94, p73], Stroustrup explains functions marked as virtual use "the Simula and C++ term for 'may be redefined later in a class derived from this one'". This avoids a huge switch statement choosing what to do at run time for a specific 'type', perhaps indicated with a flag. Any new types need to be added to the switch statement, increasing compile times and potentially introducing bugs. Stroustrup explains he adopted the Simula inheritance mechanism to avoid these problems. Simula hails from the 1960s, and the inheritance model along with subclasses may have been introduced in Simula67 [Wikipedia]. Be aware that subclassing, having a sub and super class, or base and derived as Stroustrup re-dubbed them, is different to virtual functions. [op cit, p 49] "Even without virtual functions, derived classes in C with Classes were useful for building new data structures out of old ones." I half wonder why we use the `virtual` and `override` keywords. The base class can have an implementation, so `abstract` would be the wrong word. We indicate that with `= 0 {};` well, the curly braces are only needed if we want to implement the abstract function.

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.

Without the braces, we have a pure virtual function, which is non-functional and will crash if it's ever called. Don't tell non-tech people. Their heads will explode.

Now, virtual functions are one thing, but people who know other OO languages often laugh at C++'s multiple inheritance model, and the use of a virtual base class. Language bashing often springs from not fully understanding a different paradigm. Rein yourself in if you notice you're doing this. Nonetheless, some ways of coding are less than ideal. Structured programming offered a grand improvement over jumping around between various lines of code. Though it is possible to warp your head into code laced with `goto` statements, I suspect most programmers would say it is OK to bash this way of coding. Dijkstra's famous 'Go to considered harmful' paper [Dijkstra68] is probably legendary by now, and inspires many similar talk titles. One thing we do all seem to agree on.

The pandemic has forced many things into cyberspace that used to be face to face. People are discussing the pros and cons of virtual meetups and conferences. Having not spoken at one, I can appreciate it must be odd to not have the visual or auditory feedback of attendees. People will type questions or share great quotes on social media, but it must be impossible to follow what's happening while talking into a laptop and wondering if anyone can even see or hear you. Alex Chan recently offered some advice for presenters:

> Virtual presenters should ALWAYS wear high-contrast lipstick. I'm sick of seeing presenters whose lips are barely distinguishable from their face.
>
> It makes you look even more fabulous than usual.
>
> 1 The extra contrast makes it easier for anybody who relies on lip-reading. [Chan20]

This kicked off a small discussion about the history of television, or even older, black and white films. Make-up was, and still is, used to great visual effect, and also to avoid distractions like shiny skin and so on. Sound engineering is also a full on-technical discipline, to adapt and change the real sound, making it clearer, better, or more dramatic. Even a live stage performance, for example at a real metal festival, has lighting to emphasise cool stuff, often flames, loud noises, costumes and a sound desk for reasons. Good reasons. A 'real' live performance, would be unplugged, no make-up, no lighting. You could argue it shouldn't involve any kind of 'man-made' instrument. If you don't agree, consider for a moment the source of the word virtual. It ties in with the idea of possessing certain virtues. OK, that's not so helpful. What is a virtue? Somewhere between potency or effectiveness, and manliness [Etymology-1]. If you follow the latter meaning, not only do you get the idea of 'man-made', but you get 'vir' or 'wiro' or even 'were' as a root word [Etymology-2]. A werewolf is a virtual or man wolf. I'm happy to leave the 'man' aside as perhaps meaning 'human' in this case.

Where does this leave us? Virtual reality is created by humans, but therefore has virtue. Virtual reality is no less real than reality itself. It comes in many flavours, for example sometimes we talk of augmented reality instead. Sometimes virtual and real aren't opposite. If something in tech is described as real-time, that distinguishes it from a lag or polled snapshot, rather than virtual time, whatsoever that might be. Of course, time and space are relative, so talking of 'real' time, as though there is One True Time that we can all agree on, reveals a lack of understanding of Relativity. Furthermore, many real-time operating systems or loggers are more 'near-real-time' than actual real time. This starts to beg the question, what is real anyway? We use the word carelessly, and try ideas like 'Actually existing, things… genuine'. As opposed to fake? I am told, 'The meaning 'genuine' is recorded from 1550s; the sense of 'unaffected, no-nonsense' is from 1847.' [Etymology-3] I could pull further on the history of each of these words, but copying from dictionaries is even further from an editorial than my usual excuses.

Perhaps we should consider fakes for a bit. I recently listened to a Radio 4 programme, called 'Re-enactment radio' [BBC]. Antonia Quirke and guests discuss whether scenes in movies are plausible or even possible. This time they discussed fight scenes and computers in films. Unrealistic super-hero style fight scenes got a bashing. Most unfair, to my mind. I like tightly choreographed unrealistic fight scenes. If I want to see realistic fight scenes I could go into town on a Friday night and watch the results of too much alcohol, well, could have done were it not for the virus. Swordfish [IMDb-2], some kind of covert counter-terrorist hacking story I have never seen, was then dismantled. The geeky expert on the radio called out implausible hacking into a government system, using bad C code that does not compile, and flashing 'Access Denied' messages culminating in finally managing to hack into a directory and list the contents, which included a customer satisfaction survey. Not what you'd expect to find on a government IT system. Do any of you pause a film when you see code in the background and try to figure out what language it is and if it's correct? If not, never watch a film with me.

It's easier to spot fakes when you are knowledgeable on a subject. However, this isn't fool-proof. Perhaps this begs an even more important question. Are you sure you are real, or anything is real? Are you sure you aren't a computer program, living in cyberspace? Are you living in the Matrix? You can't prove anything, all you can do is wonder and consider. You can try to be genuine, or virtuous. Don't bash virtual goings on, but do consider if you have an appropriate lip-stick for you next 'live' gig. Keep it real, as they used to say.

References

[BBC] Re-enactment Radio: https://www.bbc.co.uk/programmes/m000dk15

[Chan20] Alex Chan, tweeted 4 September 2020, https://twitter.com/alexwlchan/status/1301793743603929088

[Dijkstra68] Dijkstra, Edsger W. (March 1968). 'Letters to the editor: Go to statement considered harmful' *Communications of the ACM*. 11 (3): 147–148

[Etymology-1] Virtual: https://www.etymonline.com/word/virtual

[Etymology-2] Virtue: https://www.etymonline.com/word/virtue?ref=etymonline_crossreference

[Etymology-3] Real: https://www.etymonline.com/word/real

[IMDb-1] *Predator* (1987) https://www.imdb.com/title/tt0093773/

[IMDb-2] *Swordfish* (2001) https://www.imdb.com/title/tt0244244/

[Stroustrup94] Bjarne Stroustrup (1994) *The Design and Evolution of C++* published by Addison Wesley

[Waugh15] Rob Waugh (2015) 'Predator' becomes reality as scientists unveil a real camouflage cloak, published in https://metro.co.uk/2015/06/16/predator-becomes-reality-as-scientists-unveil-a-real-camouflage-cloak-5249172/

[Wikipedia] Simula: https://en.wikipedia.org/wiki/Simula

poly::vector – A Vector for Polymorphic Objects

Heterogeneous vectors can be slow. Janky Ferenc introduces a sequential container for storing polymorphic objects in C++.

In the Object-Oriented Programming paradigm, dealing with collections of polymorphic types is a recurring programming pattern. Heterogeneous collections storage cannot always guarantee co-locating objects, resulting in access penalties on modern CPU hardware where memory caching is utilized. This paper describes a container that has (on average) better access performance when storing polymorphic objects than other C++ Standard Template Library (STL) based variants. It achieves this by structurally enforcing the locality of references and by reducing the number of the total allocation count when a unique ownership model is desired.

Introduction and motivation

One of the most recurring patterns with the object-oriented software (OOP) design paradigm is ownership and management of ownership. In order to exploit the benefits of the design principles, the use of interface, and implementation classes are required. In the C++ language, dynamic dispatch is only applicable when a particular virtual function is called through a pointer or reference to an interface class. When pointers are involved in C++, the most frequently arising question is ownership: who owns the object, i.e., which part of the code will free up resources associated with an object?

There are multiple working solutions for this problem: smart pointers for expressing unique (`std::unique_ptr<T>`) and shared ownership (`std::shared_ptr<T>`) or the `gsl::owner<T>` from GSL libraries for marking pointers as owners of the resource while treating all others as non-owners. While the former is an active way of having a handle object through which to access the desired object and also of invoking the destructor and freeing up the allocated memory on the handle object's destruction, the latter is more like an annotation. Static analyzers can flag potential leaks and undefined behaviours related to resource management based on these annotations.

If the objective is to have a collection of polymorphic objects whose lifetime is associated with the containing data structure, the standard solution is to use one of the standard containers parameterized with a smart pointer of the interface type, e.g: `std::vector<std::unique_ptr<MyInterface>>`. That solution fulfils the requirements of resource management. However, if modern CPU architecture is considered – where locality of references is the key driver of performance – it might be sub-optimal as, typically, the memory layout will look as illustrated in Figure 1.

On most modern CPU architecture that uses out-of-order execution, overall performance is mostly affected by how the CPU cache is utilized. Assuming that it is more important to ensure the locality of references than optimizing asymptotic complexity – if the problem size is below a certain threshold where the former limit dominates performance – that locality must somehow be enforced structurally to squeeze out maximum performance from both software and hardware [Maness18]. In managed languages, there is no direct control available for the application programmer to manually control allocation layout; however, with such a system programming language as C++, this can be addressed as well – pun intended.

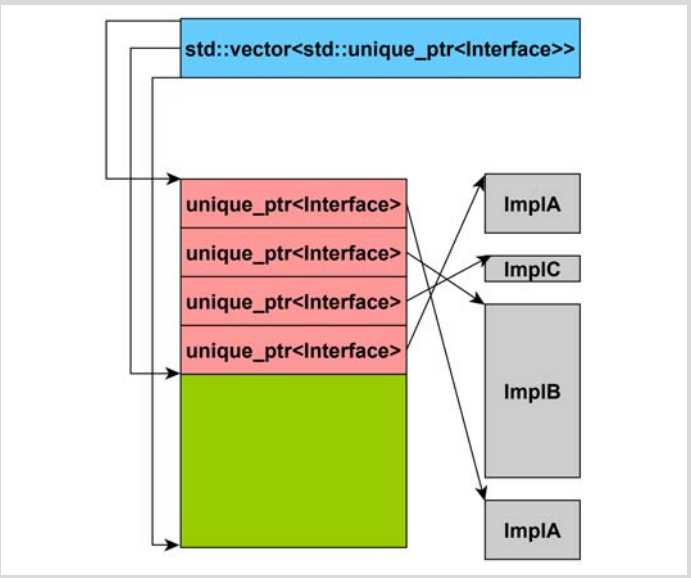

Figure 1

Nevertheless, such a data structure does not exist in the C++ Standard Library. While the Standard Template Library (STL) is generic enough for locality to be improved by using custom allocators, and C++17's recent addition of `std::pmr::monotonic_buffer_resource` makes it easier than ever. The monotonic resource's only problem is that if the assigned memory resource is exhausted, it will fall back to the upstream allocator, and the locality is not guaranteed once again. Moreover, if we examine the total allocation count when using e.g: `std::pmr::vector<unique_ptr<Interface>>`, it will be still higher than expected (including the object allocation as well).

The purpose of this article is to introduce a container that is specifically tailored for the use-case described earlier, structurally providing locality of references and keeping the total allocation count (when the data structure requests a new chunk of memory) at a minimum without the need for custom allocators. Furthermore, this container must not be comparably worse than the standard alternative in the best-case scenario while also being significantly better in the worst-case scenario. The best-case scenario means that most allocations happen successively, and in contrast, the worst-case means when random allocations happen with random longevity. Given this data structure, the memory layout would be

Janky Ferenc Nándor Ferenc received an MSc in Electrical Engineering from BME, Budapest, in 2013. He has since worked for various telecommunication companies, and is currently working as a C++ software developer for an international corporate bank. His main areas of interest are C++ programming, network protocols, FPGA programming and software development. Ferenc is a member of the SmartComLab at BME TMIT. He can be reached at mailto:janky@tmit.bme.hu

the static type of the object inserted must be known at compile time at the point of insertion

guaranteed to look something like that illustrated by Figure 2. The implementation and the benchmarks can be obtained from [Janky18].

Design

While STL already presents a solution for dynamic memory management abstraction in the form of Allocators – an application of policy-based class design [Alexandrescu11] – the key for managing/storing objects in this fashion is cloning/moving. To address the other problem of relocating objects through their interface reference, another concept has to be introduced in the form of a Cloning Policy. This can be described as a `concept`, as seen in Listing 1.

The cloning policy must be able to clone – move if supported – objects around in memory through a base class pointer. The new additions to the typical class hierarchy can be identified by the small white star (★) in Figure 3. The other component is the element pointer, which can be thought of as a decentralized smart pointer as the smart pointer operation is really realized by two collaborating classes: the vector and the element pointer itself, with the vector managing the lifetime while the access is provided through the element pointer.

Two archetypes of cloning policies have been designed: `delegate_cloning_policy` and `no_cloning_policy`. The former is suitable for the most common use cases where the derived classes are regular in terms of copy/move. As one might assume, this cloning policy captures the method of cloning/moving at the time of insertion to the container. That also means the static type of the object inserted must

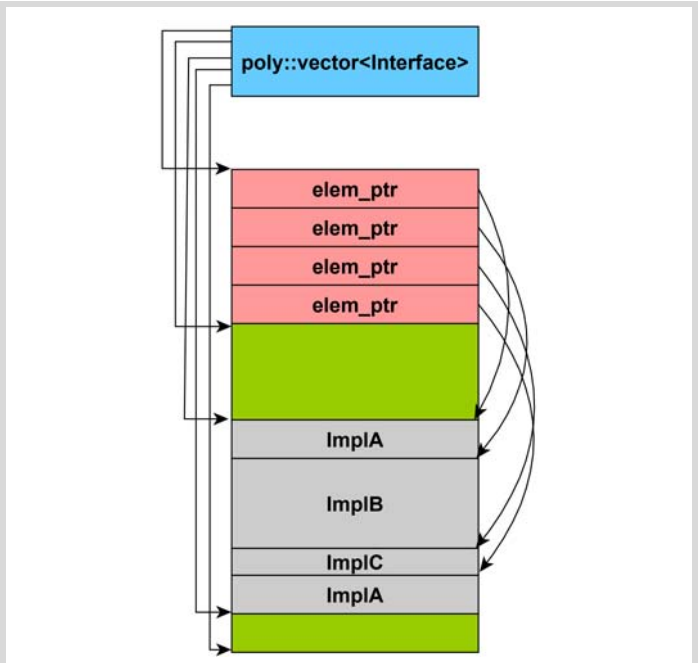

Figure 2

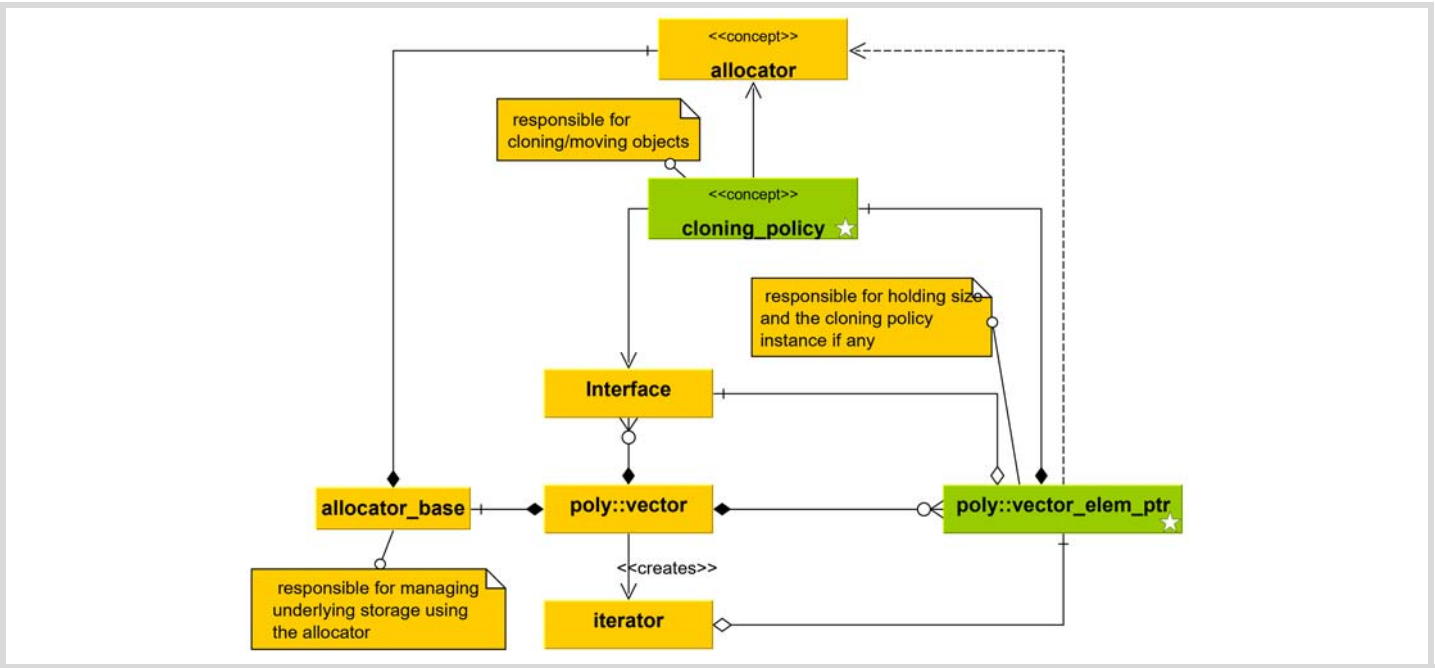

Figure 3

```
namespace poly
{
template <typename T>
struct type_tag
{
  using type = T;
};
} // namespace poly
using namespace poly ;
using namespace std;

template <typename InterfaceT, typename Allocator>
constexpr auto AllocatorPointerMatch =
  is_same_v < InterfaceT ,
  typename pointer_traits<
  typename allocator_traits < Allocator >::
  pointer >:: element_type >;

template <typename T, typename InterfaceT,
  typename Allocator >
concept HasClone = requires (T cp , Allocator a)
{
  {
    cp. clone (a,
      declval < typename allocator_traits
              < Allocator >::pointer >(),
      declval < typename allocator_traits
              < Allocator >::void_pointer >() )
  } -> same_as < typename allocator_traits
               < Allocator >:: pointer >;
};
template <typename T, typename InterfaceT,
          typename Allocator >
concept HasMove =
  is_nothrow_move_constructible_v <T> &&
  is_nothrow_move_assignable_v <T> &&
  requires (T cp , Allocator a)
  {
    {
      cp. move (a,
        declval < typename allocator_traits
                < Allocator >::pointer >(),
        declval < typename allocator_traits
                < Allocator >::void_pointer >() )
    } -> same_as < typename allocator_traits
                 < Allocator >:: pointer >;
  };

template <typename T, typename I, typename A,
  typename Derived >
concept CloningPolicy =
  AllocatorPointerMatch <I, A> &&
  is_nothrow_constructible_v <T> &&
  is_nothrow_copy_constructible_v <T> &&
  is_nothrow_copy_assignable_v <T> &&
  copyable <T> &&
  ( constructible_from <T, type_tag < Derived >>
    || default_initializable <T >) &&
  ( HasClone <T, I, A> || HasMove <T, I, A >);
```

Listing 1

be known at compile time, at the point of insertion. The latter does not allow any copy/move of an object, i.e., if the container capacity is exhausted, then a `std::bad_alloc` exception is raised.

Since `cloning_policy` is a concept – as denoted in Figure 3 – the user can supply a type that suits the actual class hierarchy whose instances are stored in the container. For example, if there is already a *Clone* virtual member function declared in the interface, a policy class can be written quickly to use that function when the container must copy the objects to a new location, e.g., because of a resize.

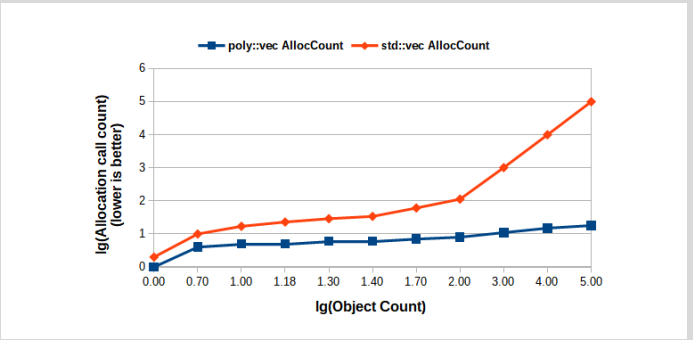

Figure 4

The classic *size* and *capacity* `concept` has to be augmented in this container: here, an average *size* is considered as size, which is the total amount of memory occupied by the objects stored in the container divided by the number of objects. The container's exponential growth is also calculated based on the average size (including the to-be inserted element in the average).

Besides these, the aim was to provide the same level of exception safety and API as one would expect from the implementation based on `std::vector<std::unique_ptr<Interface>>`.

Evaluation and measurements

Time-based measurements are not trivial to carry out during software-benchmarking and when using a non-realtime OS. For these scenarios, the measurement metric is the overall execution wall clock time. This is provided by the OS through a high resolution clock – for specific operations over a given object count that is processed through the containers [Reich18]. For the evaluation, a demo `Interface` class has been defined with two distinct and different sized implementation classes. The vectors under test were populated randomly from these two types. To minimize the variance of the measurements, the concept of static thread affinity (a.k.a thread pinning) was used hand-in-hand with setting the highest priority (smallest *nice* value) for the benchmark process. This way, the process will not be scheduled away that much, and most of the cache trashing occurs because of the benchmark program itself and not as a result of the rescheduling.

The evaluation of the container was based on three sets of measurement scenarios that have been defined as the following:

- total number of allocation count: how many times did the program allocate memory from the runtime environment – the smaller the value, the better;
- best-case scenario (when the objects are populated into the vector in a successive manner without any in-between allocations from other places): benchmark the sequential access time – the smaller the value, the better;
- worst-case scenario (where the vector is populated with objects while allocations are happening from other places, resulting in each and every object being on a separate page): benchmark the sequential access time – a smaller value is better (NB: this is simply emulated by allocating a page for an object and constructing it there).

Figure 4 shows the total allocation count, while Figure 5 the allocated size of memory, including the allocations made as a result of storing the object on a heap managed by a smart pointer handle. The data has been plotted using a log-log scale for better clarity. Figure 4 and Figure 5 should be interpreted together, e.g., for 1000 objects (at coordinate 3 on the horizontal axis) of randomly chosen types from the demo class hierarchy, there were a total of 11 calls to the allocator's allocate member function with the total size of 66006 bytes, while for the `std::vec` alternative the total allocation calls were 1018 with the total size of 24812 bytes – on average. As one can see, the total allocation count is smaller for `poly::vector`, which is a consequence of the omitted heap allocation when the objects are instantiated, while the total size is slightly greater for `poly::vector`. This is because that `poly::vector` is growing its

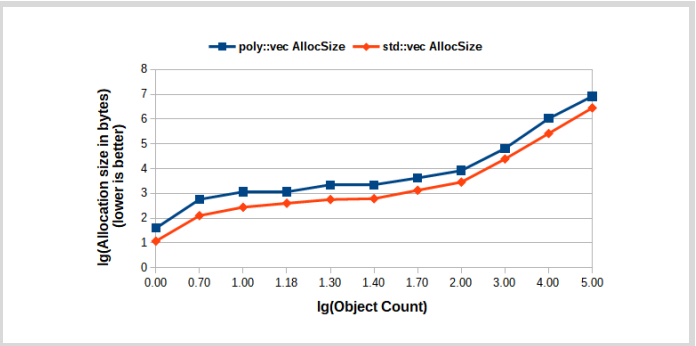

Figure 5

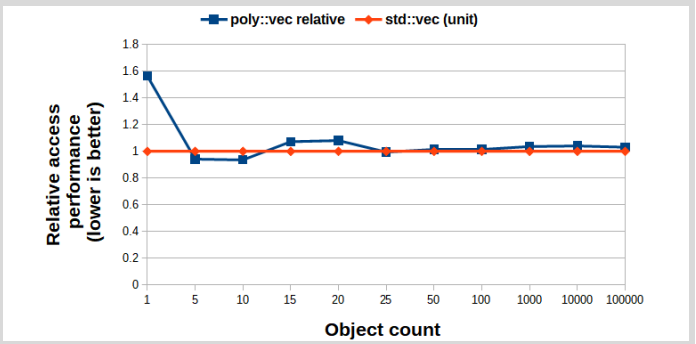

Figure 6

capacity exponentially – based on the average size that is a weighted average of all stored objects' size. This trade-off for ensuring the locality incurs somewhat higher memory utilization. However, the benefit of a reduced allocation count is obvious. Furthermore, it is even more significant when allocators from the *pmr* namespace are used, as the *allocate* call there will not be inlined normally. There is no devirtualization, as it is a dynamically dispatched call. This adds more penalty if there are more allocations made than necessary.

The measurement data that was used to generate Figure 6 and Figure 7 is based on average values. Multiple measurements were carried out for the same object count, and the statistical difference was determined by using two-tail Student T-test with unequal variances, $\alpha = 0{:}01$. For Figure 6 – which shows the averages for the best-case scenario based on the raw data

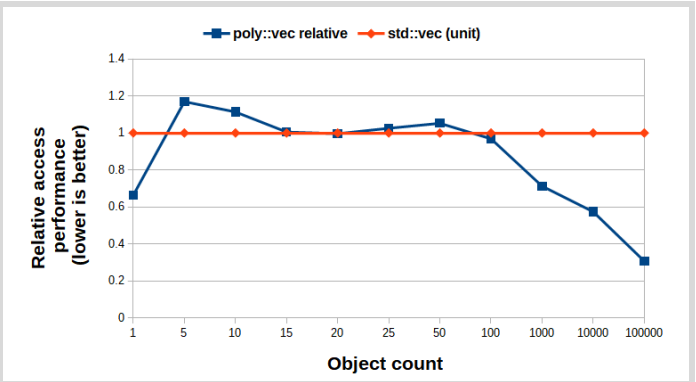

Figure 7

at the key points – there is no statistical significance of the differences between the averages. Therefore we can say that, most probably, `poly::vector` is not worse than the `std::vector` based alternative in this aspect. In the worst-case scenario – illustrated by Figure 7 – the raw data showed that until the object count reaches 100, the difference is statistically insignificant, and also showed that for a greater object count, `poly::vector` outperforms the STL-based alternative in terms of sequential access performance. (NB.: For the small object count measurements, the variance was so significant and also the timings were inaccurate that no real consequence can be deduced from those data points.)

Another important aspect – yet less tangible in terms of performance – is the syntactic verbosity of `poly::vector` compared to `std::vector`. Even though it has no runtime impact, it is still much more convenient to express ideas directly in code. As an example: if the programmer wishes to place an object of polymorphic type into a container, currently a smart pointer has to be created, memory to be allocated, the object to be constructed and assigned to the smart pointer handle instance for memory and lifetime management, then inserted into the container itself. With `poly::vector`, this is not the case: the intention is expressed directly. This argument is analogous to the `for` loop versus STL range-based algorithms. While each has its place, the intention and the business logic are still more clearly communicated using the latter.

Conclusion

In this paper, a generic container has been described that is tailored for storing polymorphic instances derived from a well-known interface. Due to the underlying memory and layout management, locality of references is enforced structurally, which results in increased sequential access performance with greater object counts, while also reducing the total number of allocation count which could also be beneficial from performance perspective. The trade-off for achieving this is an increased memory utilization, as the container maintains capacity not just for the *objects handles* but also for the yet to be stored objects based on an average size computation. It has also been shown that with the best-case allocation scheme the access performance is comparable to the standard based alternative.

In summary this container can be used as a drop in replacement for `std::vector<std::unique_ptr<InterfaceType>>` pattern in high performance applications that use OOP for abstraction but still want to eliminate the penalties due to memory fragmentation. ■

References

[Alexandrescu11] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*, Chapter 1. Addison-Wesley, 2011.

[Janky18] Ferenc Nandor Janky. poly::vector github repository. https://github.com/fecjanky/poly_vector, 2018. Accessed: 2020-09-18.

[Maness18] Wesley Maness and Richard Reich (2018) 'Cache-line aware data structures' in *Overload* 146, available at: https://accu.org/journals/overload/26/146/maness_2535/

[Reich18] Richard Reich and Wesley Maness (2018) 'Measuring throughput and the impact of cache-line awareness' in *Overload* 148, available at: https://accu.org/journals/overload/26/148/reich_2585/

Kafka Acks Explained

Kafka's configuration can be confusing. Slanislav Kozlovski helps us visualise this most misunderstood configuration setting.

Having worked with Kafka for almost two years now, there are two approaches to whose interaction I've seen to be ubiquitously confused. Those two configs are `acks` and `min.insync.replicas` – and how they interplay with each other.

This article aims to be a handy reference which clears the confusion through the help of some illustrations.

Replication

To best understand these configs, it's useful to remind ourselves of Kafka's replication protocol.

I'm assuming you're already familiar with Kafka – if you aren't, feel free to check out my 'A Thorough Introduction to Apache Kafka' article [Kozlovski20].

For each partition, there exists one leader broker and *n* follower brokers. The config which controls how many such brokers (1+*N*) exist is `replication.factor`. That's the total amount of times the data inside a single partition is replicated across the cluster. The default and typical recommendation is 3 (see Figure 1).

Producer clients only write to the leader broker – the followers asynchronously replicate the data. Now, because of the messy world of distributed systems, we need a way to tell whether these followers are managing to keep up with the leader – do they have the latest data written to the leader?

In-sync replicas

An *in-sync replica* (ISR) is a broker that has the latest data for a given partition. A *leader* is always an in-sync replica. A *follower* is an in-sync replica only if it has fully caught up to the partition it's following. In other words, it can't be behind on the latest records for a given partition.

If a follower broker falls behind the latest data for a partition, we no longer count it as an in-sync replica. See Figure 2, which shows that Broker 3 is behind (out of sync).

Note that the way we determine whether a replica is in-sync or not is a bit more nuanced – it's not as simple as 'Does the broker have the latest record?' Discussing that is outside the scope of this article. For now, trust me that red brokers with snails on them are out of sync.

Acknowledgements

The `acks` setting is a client (producer) configuration. It denotes the number of brokers that must receive the record before we consider the write as successful. It support three values – `0`, `1`, and `all`.

'acks=0'

With a value of `0`, the producer won't even wait for a response from the broker. It immediately considers the write successful the moment the record is sent out. (See Figure 3: The producer doesn't even wait for a response. The message is acknowledged!)

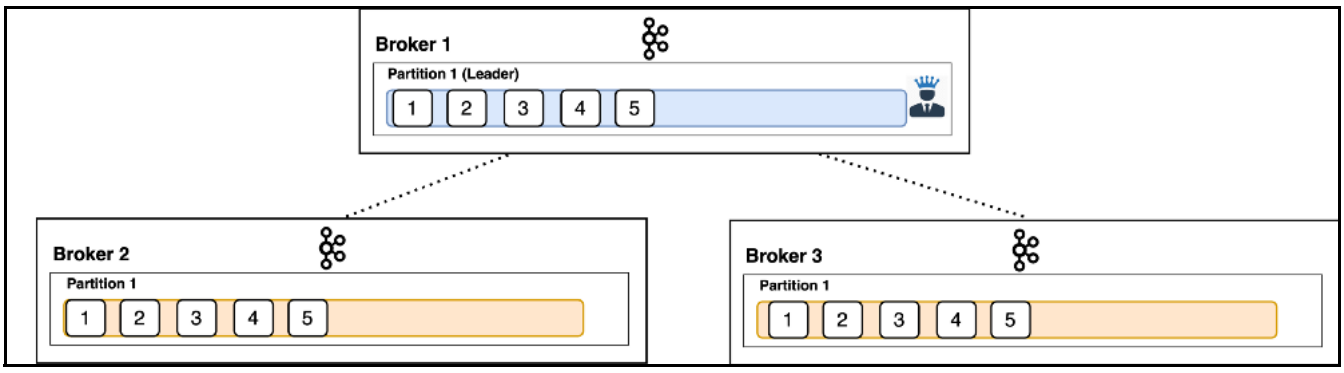

Figure 1

Stanislav Kozlovski Stanislav began his programming career racing through some coding academies and bootcamps, where he aced all of his courses and began work at SumUp, a German fintech company aiming to become the first global card acceptance brand. He was later recruited into Confluent, a company offering a hosted solution and enterprise products around Apache Kafka. Contact him on Twitter, where he's @BdKozlovski or at Stanislav_Kozlovski@outlook.com

Apache Kafka

Apache Kafka is a battle-tested event streaming platform that allows you to implement end-to-end streaming use cases. It allows users to publish (write) and subscribe to (read) streams of events, store them durably and reliably, and process these stream of events as they occur or retrospectively.

Kafka is a distributed, highly scalable, elastic, fault-tolerant and secure system used by more than one-third of Fortune 500 companies.

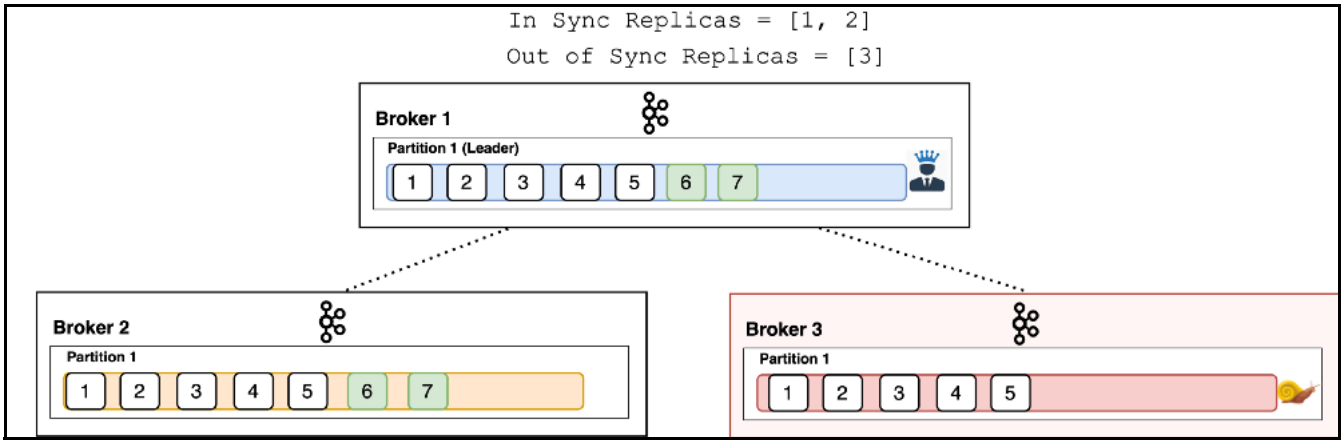

Figure 2

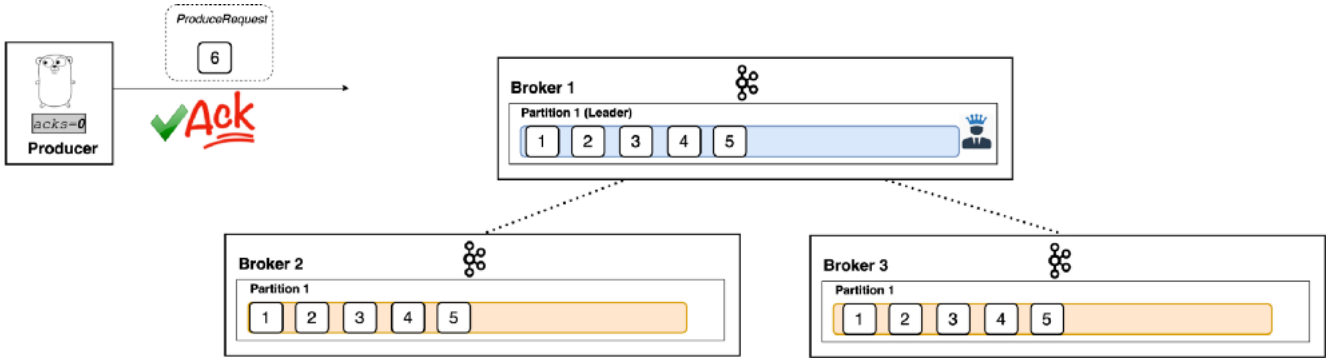

Figure 3

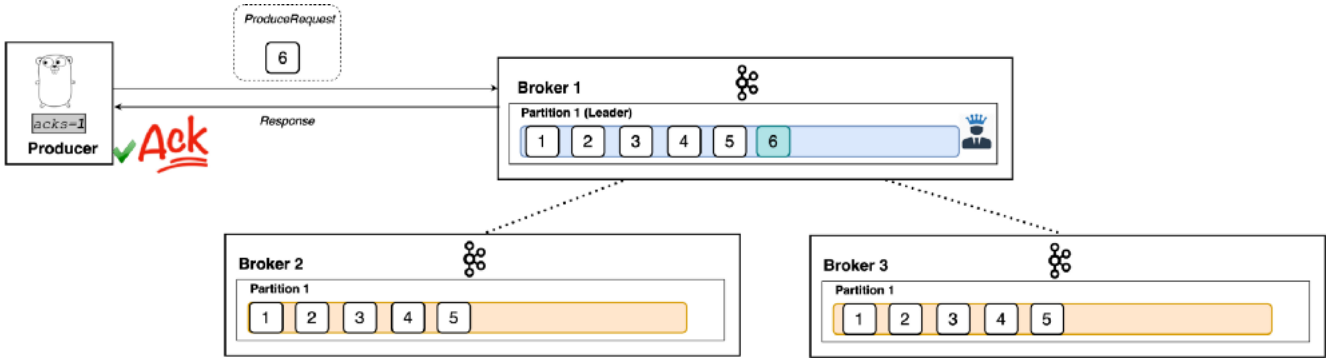

Figure 4

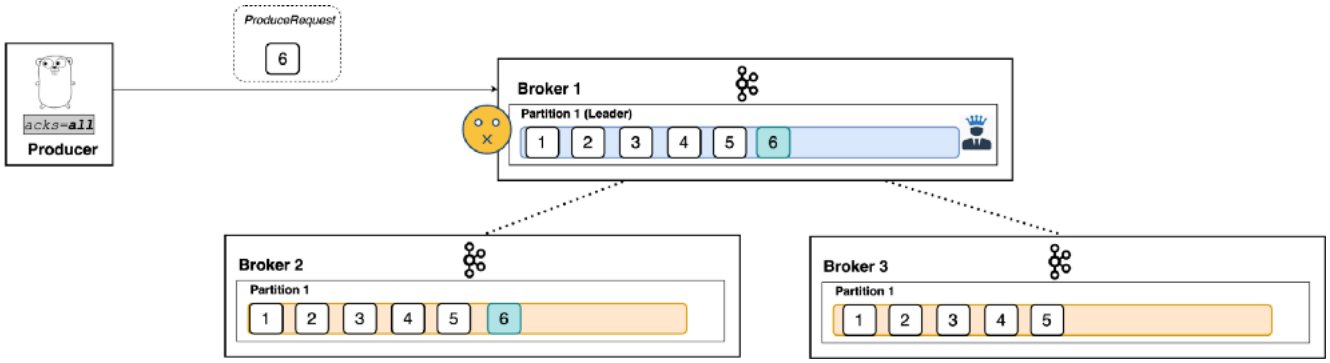

Figure 5

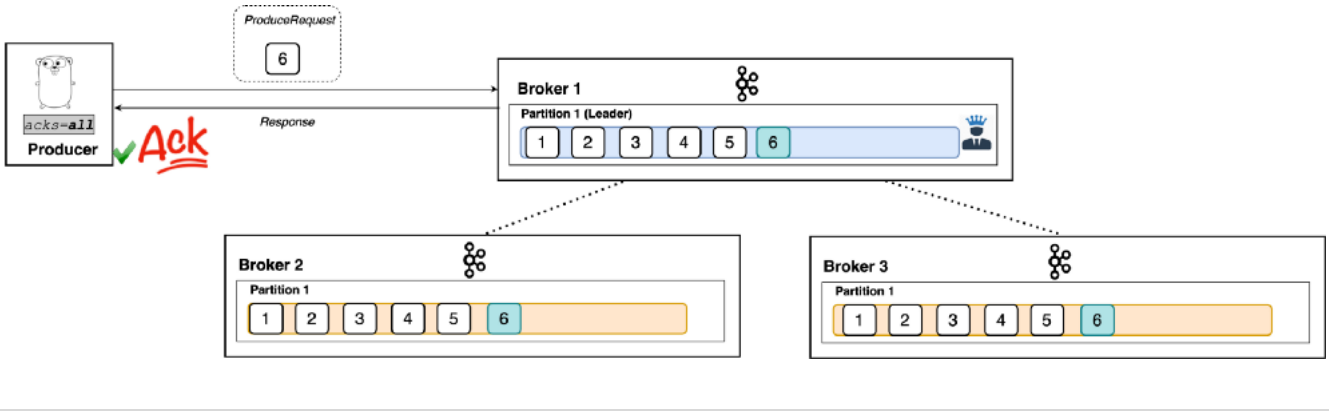

Figure 6

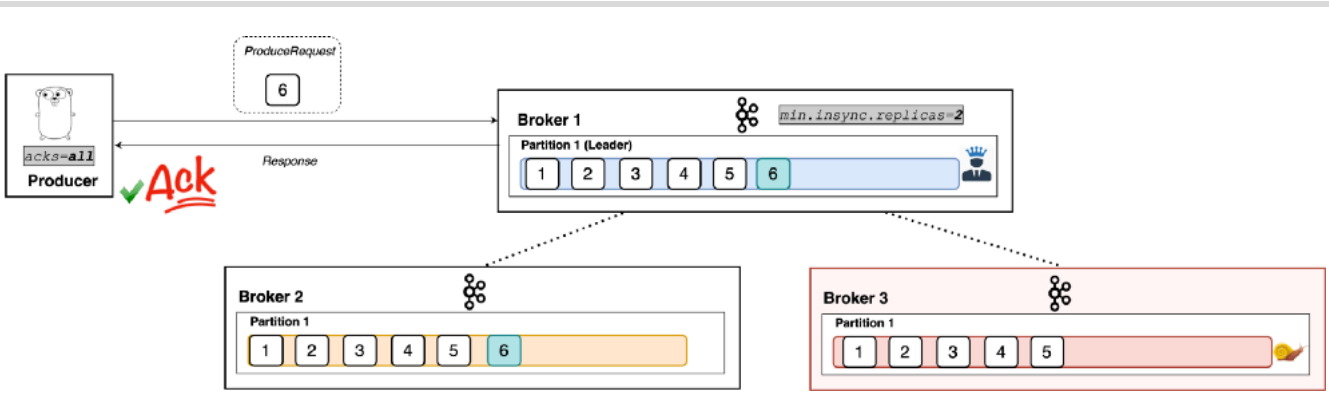

Figure 7

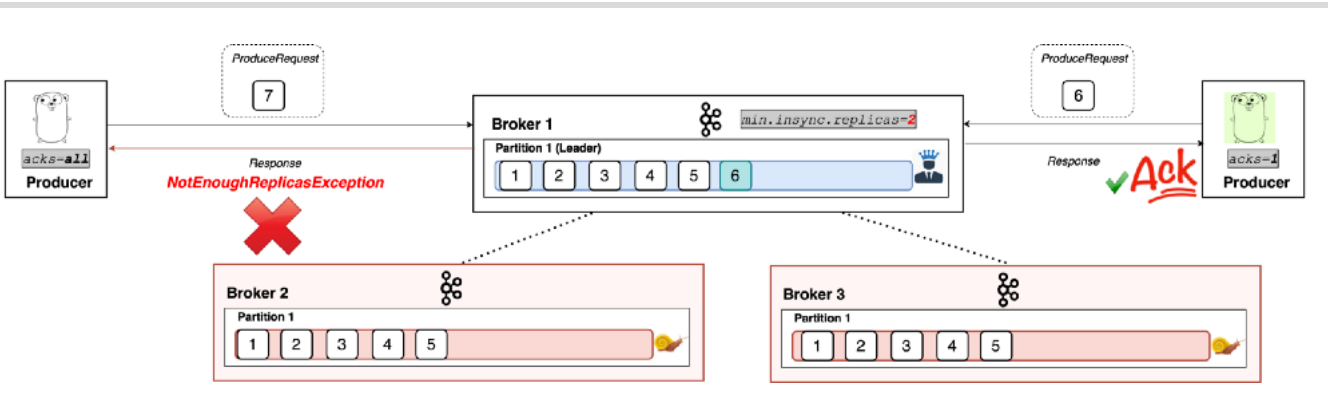

Figure 8

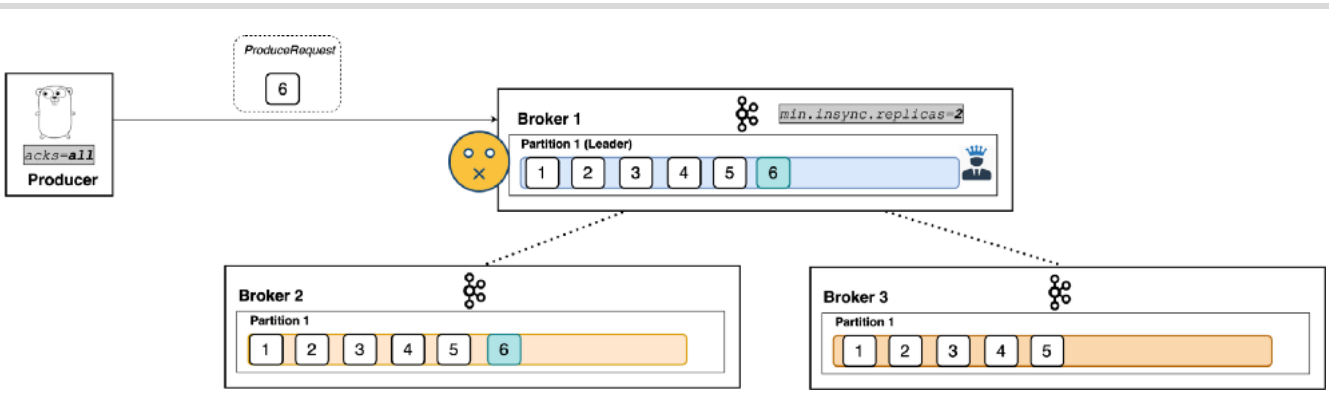

Figure 9

'acks=1'

With a setting of `1`, the producer will consider the write successful when the leader receives the record. The leader broker will know to immediately respond the moment it receives the record and not wait any longer. (See Figure 4: The producer waits for a response. Once it receives it, the message is acknowledged. The broker immediately responds once it receives the record. The followers asynchronously replicate the new record.)

'acks=all'

When set to `all`, the producer will consider the write successful when all of the in-sync replicas receive the record. This is achieved by the leader broker being smart as to when it responds to the request – it'll send back a response once all the in-sync replicas receive the record themselves. (See Figure 5: Not so fast! Broker 3 still hasn't received the record.)

Like I said, the leader broker knows when to respond to a producer that uses `acks=all`. (See Figure 6: Ah, there we go!)

Acks's utility

As you can tell, the `acks` setting is a good way to configure your preferred trade-off between durability guarantees and performance.

If you'd like to be sure your records are nice and safe – configure your acks to `all`.

If you value latency and throughput over sleeping well at night, set a low threshold of `0`. You may have a greater chance of losing messages, but you inherently have better latency and throughput.

Minimum in-sync replica

There's one thing missing with the `acks=all` configuration in isolation.

If the leader responds when all the in-sync replicas have received the write, what happens when the leader is the only in-sync replica? Wouldn't that be equivalent to setting `acks=1`?

This is where `min.insync.replicas` starts to shine!

`min.insync.replicas` is a config on the broker that denotes the minimum number of in-sync replicas required to exist for a broker to allow `acks=all` requests. That is, all requests with `acks=all` won't be processed and receive an error response if the number of in-sync replicas is below the configured minimum amount. It acts as a sort of gatekeeper to ensure scenarios like the one described above can't happen. (See Figure 7: Broker 3 is out of sync).

As shown, `min.insync.replicas=X` allows `acks=all` requests to continue to work when at least *x* replicas of the partition are in sync. Here, we saw an example with two replicas.

But if we go below that value of in-sync replicas, the producer will start receiving exceptions. (See Figure 8: Brokers 2 and 3 are out of sync.)

As you can see, producers with `acks=all` can't write to the partition successfully during such a situation. Note, however, that producers with `acks=0` or `acks=1` continue to work just fine.

Caveat

A common misconception is that `min.insync.replicas` denotes how many replicas need to receive the record in order for the leader to respond to the producer. That's not true – the config is the *minimum* number of in-sync replicas required to exist in order for the request to be processed. That is, if there are three in-sync replicas and `min.insync.replicas=2`, the leader will respond only when all three replicas have the record. (See Figure 9: Broker 3 is an in-sync replica. The leader can't respond yet because broker 3 hasn't received the write.)

Summary

And that's all there is to it! Simple once visualized – isn't it?

To recap, the `acks` and `min.insync.replicas` settings are what let you configure the preferred durability requirements for writes in your Kafka cluster.

- `acks=0` – the write is considered successful the moment the request is sent out. No need to wait for a response.
- `acks=1` – the leader must receive the record and respond before the write is considered successful.
- `acks=all` – all online in sync replicas must receive the write. If there are less than `min.insync.replicas` online, then the write won't be processed. ■

Further Reading

Kafka is a complex distributed system, so there's a lot more to learn about!

Here are some resources I can recommend as a follow-up:

- Kafka consumer data-access semantics (https://www.confluent.io/blog/apache-kafka-data-access-semantics-consumers-and-membership/) – A more in-depth blog of mine that goes over how consumers achieve durability, consistency, and availability.
- Kafka controller (https://medium.com/@stanislavkozlovski/apache-kafkas-distributed-system-firefighter-the-controller-broker-1afca1eae302) – Another in-depth post of mine where we dive into how coordination between brokers works. It explains what makes a replica out of sync (the nuance I alluded to earlier).
- '99th Percentile Latency at Scale with Apache Kafka' (https://www.confluent.io/blog/configure-kafka-to-minimize-latency/) – An amazing post going over Kafka performance – great tips and explanation on how to configure for low latency and high throughput.
- Kafka Summit SF 2019 videos: https://www.confluent.io/resources/kafka-summit-san-francisco-2019/
- Confluent blog (https://www.confluent.io/blog/) – a wealth of information regarding Apache Kafka
- Kafka documentation (https://kafka.apache.org/documentation/) – Great, extensive, high-quality documentation.

Kafka is actively developed – it's only growing in features and reliability due to its healthy community. To best follow its development, I'd recommend joining the mailing lists (https://kafka.apache.org/contact).

Reference

[Kozlovski20] Stanislav Kozlovski '' in Overload 159, August 2020, available at: https://accu.org/journals/overload/28/158/kozlovski/

This article was first published on Stanislav's blog on 29 March https://medium.com/better-programming/kafka-acks-explained-c0515b3b707e

Concurrency Design Patterns

Orchestrating concurrent tasks using mutexes is seldom efficient. Lucian Tadu Teodorescu investigates design patterns that help unlock concurrent performance.

If you are a reader of *Overload*, then you probably know by now that mutexes should be avoided and tasks can be a viable alternative to them. If you are not an *Overload* reader, you are missing out ☺.

In the last two articles [Teodorescu20a] [Teodorescu20b], I tried to show that using tasks instead of mutexes is more performant, is safer and they can be employed in all the places that mutexes can. Tasks are not the only alternative to mutexes, but this seems to be the most general alternative; to a large degree, one can change all programs that use mutexes to use tasks. In general, using tasks, one shifts focus from the details of implementing multithreaded applications to designing concurrent applications. And, whenever the focus is on design, we can be much better at the task at hand – design is central to the software engineering discipline.

But, as tasks are not very widespread, people may not have sufficient examples to start working with tasks instead of mutexes. This article tries to help with this by providing a series of design patterns that can help ease the adoption of task systems, and that may, at the same time, improve general concurrency design skills. Even more fundamentally, it tries to show how applications can be designed for concurrency.

Concurrency vs. parallelism

There is widespread confusion in the software industry between concurrency and parallelism. Before moving forward with the article, it's worth clarifying the distinction.

Parallelism is about running multiple things in parallel; concurrency is about ensuring that multiple things *can* run in parallel (or, more correctly, at the same time), the composition of independent processes[1]. For parallelism, one needs to have at least two CPU cores; on the other hand, concurrency can be expressed on a single core too. Having two threads doesn't imply parallelism, but it implies concurrency. So, parallelism implies concurrency, but not the other way around. To benefit from parallelism, one needs to design for concurrency.

If we look from a performance point of view, one wants parallelism, but needs to design the code for concurrency. If we look from a design point of view, then we should mostly be only concerned with concurrency. At design time, it's not clear if at run-time one will have the hardware to run things in parallel. The goal of concurrency is to structure programs, but not necessarily to make them run with more parallelism – in the best case, it is an enabler for parallelism.

See [Pike13] for a better-articulated distinction between the two concepts.

We are focusing here on the design aspects, on how to express concurrent processes, and, therefore, we will ignore parallelism for the most part.

Approach

Drawing inspiration from Christopher Alexander [Alexander77], Gamma et all. [Gamma94] popularized the idea that designing software systems can be greatly improved by using patterns. Instead of working up all the details of a software system, one can get inspiration from various design pattern to speed up the design process. In some sense, patterns are a formalization of collective experience; using this experience can greatly help the design process.

Here, we aim at leveraging patterns as a compact way of transmitting a body of experience in designing concurrent systems. Mixing and matching these patterns can help to solve a large variety of concurrency problems. Moreover, as we present some fundamental patterns, I would argue that it can help solve any concurrency problem – maybe, in some cases, not the best solution, but still a solution. The reader is strongly encouraged to see these patterns as building blocks and start playing with them to build more and more complex concurrency systems.

Because of the space constraints, we will expose a compact version for each pattern. We will say a few words about what each pattern is about, when to use it and, when appropriate, some key points. Diagrams seem to help a lot the process of reasoning about the design of a system, so we'll make sure to include a diagram for each pattern we discuss. Also, for each pattern we'll provide a short example in C++ code, using my Concore library [concore][2]. One of the benefits of using these examples is to show that, using an appropriate library, one can easily express concurrency in C++. After all, concurrency doesn't need to be one of the hardest areas in computer science.

Basic concurrency patterns

Create concurrent work

Description. Allows the creation of concurrent work; increases concurrency. Creates new tasks (typically two or more) from the existing task.

Representation. See Figure 1. We represent this pattern by multiple arrows originating from the single task, leading to new tasks. There can be more than two tasks created from one task, so there can be more than two arrows.

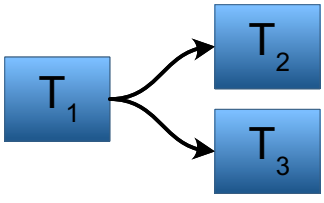

Figure 1

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. In his spare time, he is working on his own programming language and he is improving his Chuck Norris debugging skills: staring at the code until all the bugs flee in horror. You can contact him at lucteo@lucteo.ro

1. We use the term *processes* here in the same way that Hoare uses it in his seminal book [Hoare85]; it means any body of work; not to be confused with OS processes.
2. Concore is not yet a production-ready library; things may change in the future, both in terms of API and of features.

With respect to performance, is often better to over-split the work into multiple concurrent processes than to split it less than needed

```
void start() {
  initComponents();
  concore::spawn([]{ loadAssets(); });
  concore::spawn([]{
    initiliseComputationEngine (); });
}
```

Listing 1

Example. See Listing 1. In the body of the first task (`start()` function), we do some work, then we spawn two new tasks; the two tasks can be executed concurrently. For spawning the two tasks we use lambdas, as they are perfect for the job. Tasks are implemented using functors that take no arguments and return nothing. (We can build tasks that pass values around by binding values to the lambdas used to create the tasks.)

Discussion. There are multiple ways in which a task can be given to the task system; this example uses the `spawn` function. Another way to do it is to use a `global_executor` or some other type of executor. Executors, in Concore, can be used to customize how tasks are executed (they are somehow similar to the executors proposed for the C++ Standard, but, at least for the moment, they are not the same).

Key point. It's important to ensure safety for the concurrent tasks that are created; i.e., there should be no race condition bugs between the set of tasks that are created. See [Teodorescu20b] for more details. In our example, `showSplashScreen()` should not interfere with `loadAssets()`.

When to use. To increase concurrency in an application, this should be used as often as our correctness allows (and performance does not degrade). With respect to performance, is often better to over-split the work into multiple concurrent processes than to split it less than needed – it's much easier, later on, to combine work later on than to split it further. But, if we have enough tasks in the system, and we want to maximize performance, considering the results from [Teodorescu20a], we should split tasks so that the sizes of the tasks are several orders of magnitude higher than the overhead generated by the task framework.

Continuation

Description. Allows 'do X after finishing Y' workloads to be encoded. Allows decoupling between the first task and its continuation (see [Wikipedia]). Can also be used to split longer tasks into smaller ones, for performance reasons.

Representation. See Figure 2.

Example. Listing 2 shows an example of how this pattern can be used; we have an HTTP engine that makes async requests, and whenever the response comes back, it executes a continuation; the example shows just how the continuation is started, and how the top-level API can be used. One can see that the HTTP implementation is decoupled from the response handling logic; the latter is passed as a continuation to the HTTP engine.

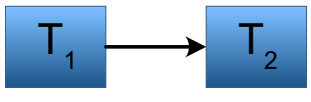

Figure 2

```
void handleResponse(HttpResponseData respData,
HandlerType handler) {
  // the work for this task: process the response
  HttpResponse resp = respData.toResponse();
  // create a continuation to handle the response
  concore::task cont{[resp = std::move(resp),
     handler] {
    handler(resp);
  }};
  concore::spawn(std::move(cont));
}
void httpAsyncCall(const char* url,
  HandlerType handler) {
  // does HTTP logic, and eventually async
  // calls handleRespnse()
}
void useHttpCode() {
  // the work to be executed as a continuation
  HandlerType handler = [](HttpResponse resp) {
    printResponse(resp);
  };
  // call the Http code asynchronously, passing the
  // continuation work
  httpAsyncCall("www.google.com", handler);
  // whenever the response comes back,
  // the above handler is called
}
```

Listing 2

Discussion. This is similar to the creation of concurrent work, but we just create one follow-up task. In this pattern, the follow-up task is always started at the end of the first task.

When to use. Mostly when we need to decouple two actions that need to be executed serially. Sometimes when we just need to break a long task into smaller tasks (which can improve performance by giving more flexibility to the task scheduler).

See also. Creating of concurrent work, serializers.

Join

Description. Allows the execution of work whenever multiple concurrent tasks are finished. The inverse of the creation of concurrent work pattern.

Representation. See Figure 3.

Example. Listing 3 shows how the problem from Listing 1 can continue; when both the two tasks are complete, a `finish_task` is started. At the end of each task, they have to notify an event object to ensure that the `fhish_task` starts at the right time.

In task-based programming, people are encouraged to minimize the number of waits in favour of creating new tasks

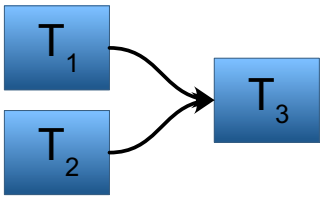

Figure 3

Discussion. The way this pattern is expressed, we create a task whenever the previous tasks are done, as opposed to somebody waiting for the tasks to be complete. In task-based programming, people are encouraged to minimize the number of waits in favour of creating new tasks.

When to use. Whenever several tasks need to complete before starting another task (e.g., they compute something that is needed for the successor task).

Fork-join

Description. This pattern is somehow a combination of the creation of concurrent work and the join pattern. But we present it here separately for its peculiar way of handing the stack and continuing the work; the thread that created the work also waits for the work to be completed, and its stack remains intact. This is actually a busy-wait. See [McCool12], [Robison14] for more details.

Representation. See Figure 4.

Example. Listing 4 shows a recursive generic algorithm that applies a functor over a range of integers. While the interval is large enough, it divides it and recursively applies the functor. It's important to notice that, even if this works with tasks, the stacks for all the recursive calls are kept alive by the `wait()` calls.

Discussion. As tasks are functions that do not take any parameters and don't return anything, the way to pass information between tasks is via

```
concore::finish_task doneTask([]{
  listenForRequests();
}, 2); // waits on 2 tasks

// Spawn 2 tasks
auto event = doneTask.event();
concore::spawn([event] {
  loadAssets();
  event.notify_done();
});
concore::spawn([event] {
  initiliseComputationEngine();
  event.notify_done();
});
// When they complete, the done task is triggered
```

Listing 3

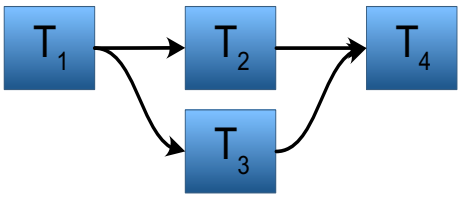

Figure 4

captured variables in the given functors/lambdas. Typically, if the stack is not available, the data passed between the tasks need to be allocated on the heap. By keeping the stack around, this pattern allows the user to avoid allocating data on the heap. It also simplifies the handling of the data (i.e., don't need to pack the data in additional structures). This can save a lot of development time if one wants to improve concurrency for a piece of code that is found at the bottom of a somehow larger callstack.

Key point. This pattern waits on the caller thread/task. But, it's important to realize that this is a busy-wait. If it cannot execute any tasks that have just forked, it will attempt to execute other tasks from the system in the hope that the forked tasks finish as soon as possible. While trying to maintain a constant throughput, this pattern may slightly damage the latency of certain operations.

When to use. Whenever the fork and the join need to happen in the same area of code, whenever we want to take advantage of the stack, or whenever it's too complex to refactor the code to use continuations and regular join patterns.

Designing with basic patterns

After describing these basic patterns, we should pause and reflect on their usage. They can be combined in a lot of ways to describe any problem that can be expressed as a direct acyclic graph. Moreover, with a little creativity (i.e., creating some helper control tasks), we can also handle arbitrary

```
template <typename F>
void conc_apply(int start, int end,
    int granularity, F f) {
  if (end - start <= granularity)
    for (int i = start; i < end; i++)
      f(i);
  else {
    int mid = start + (end - start) / 2;
    auto grp = concore::task_group::create();
    concore::spawn([=] { conc_apply(start, mid,
      granularity, f); }, grp);
    concore::spawn([=] { conc_apply(mid, end,
      granularity, f); }, grp);
    concore::wait(grp);
  }
}
```

Listing 4

After describing these basic patterns, we should pause and reflect on their usage – they can be combined in a lot of ways

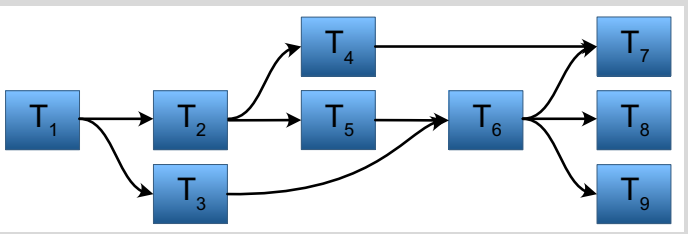

Figure 5

restrictions [Teodorescu20b]. This means that all 4 of these basic patterns can be used to implement any concurrency problem. This is a powerful design tool.

Derived patterns

Task graph

Description. Allows the expression of directed acyclic graphs of tasks directly.

Representation. See Figure 5 for an example of a task graph.

Example. Listing 5 shows an example of how one can code the graph from Figure 5. After constructing the tasks, one can set the dependencies between the tasks to match the desired graph. To start executing, one has to schedule the first task from the graph.

Discussion. The defined graph must be acyclic, otherwise, the tasks will not run. It's also worth noting that the task graph doesn't necessarily need to start with just one task; one can have graphs that have multiple starting points. This allows the modelling of much more complex flows.

When to use. Whenever the execution flow is clear upfront and/or the graph is slightly more complex.

Pipeline

Description. Allows the expression of data pipelines that can process items concurrently.

Representation. See Figure 6 for an example of a pipeline with 4 stages, 2 in order and 2 concurrent.

Example. Listing 6 shows a classic pipeline with stages for *Decode*, *Fetch*, *Execute* and *Write*. The *Decode* and *Write* stages need to run the elements in the order in which they are pushed to the pipeline, but the other two stages can be executed concurrently. For any item pushed through the pipeline, all the stage functions will be executed in order; they would all receive the shared pointer to the same data. We gain concurrency by allowing multiple items to be in the *Fetch* and *Execute* stages. The execution of the *Decode* and *Write* stages is serialized, and the items are processed in the order in which they are pushed.

Discussion. Each item that goes through a pipeline must follow a certain number of stages, in sequence. But, in some cases, several items can go through the pipeline concurrently. A pipeline can typically limit the maximum number of items that are processed concurrently. In a classical

```
std::shared_ptr<RequestData> data
  = CreateRequestData();
// create the tasks
concore::chained_task t1{[data] {
  ReadRequest(data); }};
concore::chained_task t2{[data] { Parse(data); }};
concore::chained_task t3{[data] {
  Authenticate(data); }};
concore::chained_task t4{[data] {
  StoreBeginAction(data); }};
concore::chained_task t5{[data] {
  AllocResources(data); }};
concore::chained_task t6{[data] {
  ComputeResult(data); }};
concore::chained_task t7{[data] {
  StoreEndAction(data); }};
concore::chained_task t8{[data] {
  UpdateStats(data); }};
concore::chained_task t9{[data] {
  SendResponse(data); }};
// set up dependencies
concore::add_dependencies(t1, {t2, t3});
concore::add_dependencies(t2, {t4, t5});
concore::add_dependency(t4, t7);
concore::add_dependencies({t3, t5}, t6);
concore::add_dependencies(t6, {t7, t8, t9});
// start the graph
concore::spawn(t1);
```

Listing 5

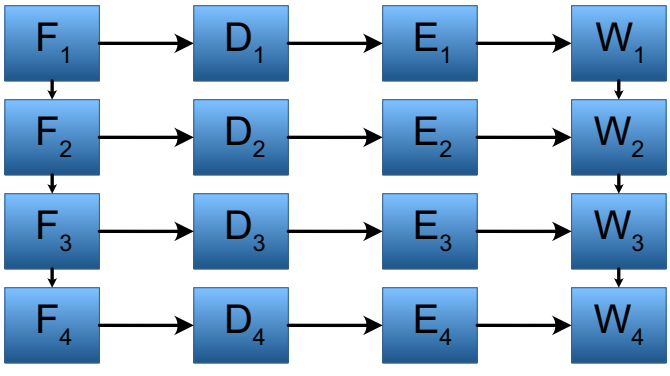

Figure 6

pipeline, processing items at any stage is ordered, but one may want to relax these restrictions. In the above example, we enforce the *Fetch* and the *Write* stages to be ordered, but we didn't impose any limit on the middle stages; the middle stages are allowed to be fully concurrent. Between an ordered restriction (first and last stages) and no restrictions at all, there is another type of restriction that one may want to use: *out-of-order serial*.

Improving concurrency is directly related to relaxing some of the constraints of the original model.

```
using LinePtr = std::shared_ptr<LineData>;
auto my_pipeline =
concore::pipeline_builder<LinePtr>()
  | concore::stage_ordering::in_order
  | [](LinePtr line) { Fetch(std::move(line)); }
  | concore::stage_ordering::concurrent
  | [](LinePtr line) { Decode(std::move(line)); }
  | [](LinePtr line) { Execute(std::move(line)); }
  | concore::stage_ordering::in_order
  | [](LinePtr line) { Write(std::move(line)); }
  | concore::pipeline_end;

for (int i = 0; i < num_lines; i++)
  my_pipeline.push(get_line(i));
```

Listing 6

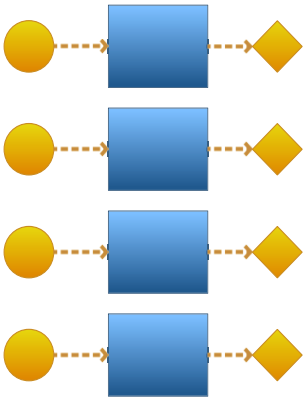

Figure 7

In this mode, the system is allowed to execute at most one task per stage, but it doesn't need to be in order.

Key point. This pattern is a great example of how to change an apparently sequential system and add concurrency to it. Improving concurrency is directly related to relaxing some of the constraints of the original model. The first constraint we drop is that we can execute a maximum of one item concurrently; it turns out that if we keep the input and the output stage ordered, most of the time we don't need this constraint. Also, if one can move most of the work in a pipeline in stages that are not fully concurrent, this can improve concurrency a lot; for example, if one spends more than half of the total time in concurrent stages of the pipeline, then given that we have enough items that flow through the pipeline, the concurrency will steadily grow.

When to use. Whenever we have a process that needs to execute sequentially some steps over some items, but some of the steps can run concurrently.

Serializers

Serializers are presented in detail in the previous article [Teodorescu20b], so we won't cover them here. The main idea that we want to stress here is that they are often design patterns in the concurrency world. In my experience, I find that using a serializer is one of the most frequent first-choices whenever expressing constraints between tasks; maybe a bit too often.

Data transformation patterns

Concurrent for

Description. Allows the concurrent execution of a body of work over a collection of items, or transforming a collection of elements with a mapping function.

Representation. See Figure 7 for a representation of a data transformation. Yellow/light circles and diamonds represent data.

Example. Listing 7 shows how one can apply a transformation to a collection of elements.

Discussion. This looks very much like a `for` structure, in which all the iterations can be executed concurrently. Because of that, it's probably the easiest form of concurrency.

When to use. Whenever the iterations of a `for` loop are independent of each other.

Concurrent reduce

Description. A concurrent version of `std::accumulate`, allowing reduction over a collection of items.

Representation. Figure 8 shows the inner tasks involved in reducing a collection of 4 elements.

Example. Listing 8 shows how one can use concurrent reduce operations to compute the total memory consumption for a vector of resources. It's assumed that getting the memory consumption of one resource is independent of getting the memory consumption for another resource.

Discussion. This is similar to a concurrent `for`, but also reduces the results obtained from all iterations into one value. The reduction operation is not

```
std::vector<int> ids = getAssetIds();
int n = ids.size();
std::vector<Asset> assets(n);
concore::conc_for(0, n, [&](int i) {
  assets[i] = prepareAsset(ids[i]); });
```

Listing 7

Adding more work is sometimes used to add concurrency to an algorithm; the hope is that even with the added work, the added concurrency will make the algorithm faster

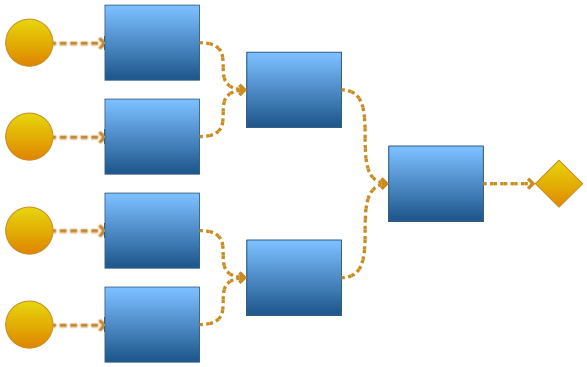

Figure 8

linearly performed as in the case of a traditional `for` loop, and therefore the reduction needs to be associative.

When to use. Whenever one needs a reduction over a collection, and the operations involved can run concurrently.

```
std::vector<Resource> res = getResources();
auto oper = [&](int prevMem, const Resource& res)
-> int {
    return prevMem + getMemoryConsumption(res);
};
auto reduce = [](int lhs, int rhs) -> int {
  return lhs + rhs; };
int totalMem = concore::conc_reduce(res.begin(),
  res.end(), 0, oper, reduce);
```

Listing 8

```
std::vector<FeatureVector> in = getInputData();
std::vector<FeatureVector> out(in.size());
auto op = [](FeatureVector lhs, FeatureVector rhs)
-> FeatureVector {
  return combineFeatures(lhs, rhs);
};
concore::conc_scan(in.begin(), in.end(),
  out.begin(), FeatureVector(), op);
```

Listing 9

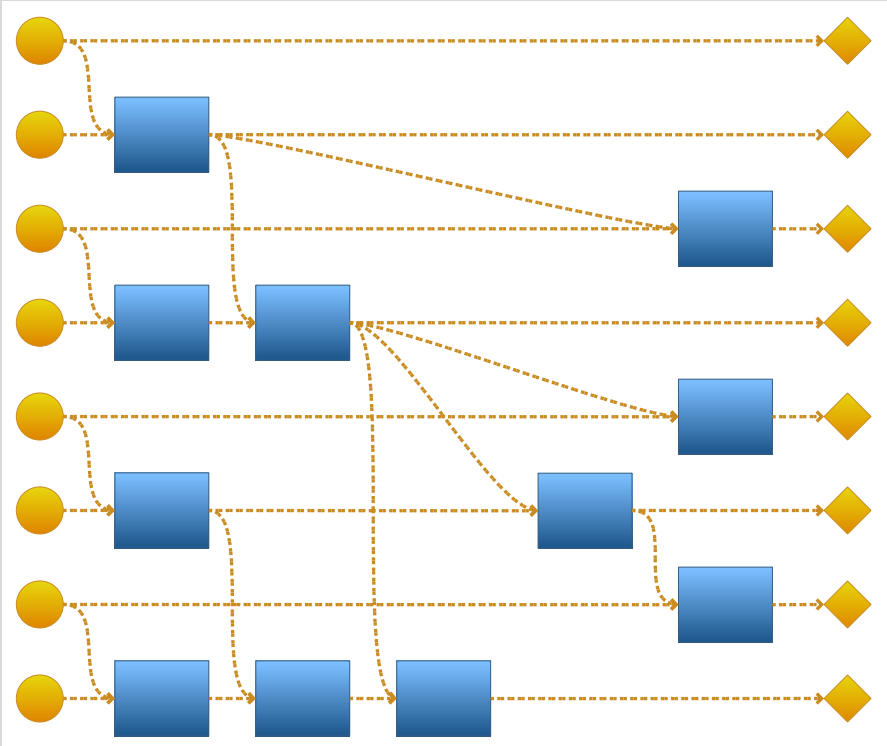

Figure 9

Concurrent scan

Description. Allows concurrent execution of loops that have dependencies of the results computed in the previous iteration; implements a concurrent version of `std::partial_sum`.

Representation. Figure 9 shows the processing needed to apply a concurrent scan over 8 elements.

Example. Listing 9 shows an example in which we have a vector of feature vectors, and we want to successively combine these, and keep all the intermediate results.

Discussion. Naively, as one needs the result of the previous iteration in the current iteration (i.e., $out_i = out_{i-1} \oplus in_i$), one may think that this cannot be done concurrently. But, if the operation that we apply is associative, then this can also be transformed into a concurrent algorithm.

Key point. The implementation of this concurrent algorithm does more work than its serial counterpart. Adding more work is sometimes used to add concurrency to an algorithm; the hope is that even with the added work, the added concurrency will make the algorithm faster.

When to use. Whenever one needs to do a *prefix sum* type of algorithm and the speed of the algorithm should be improved by running it in parallel.

there is no need to use synchronization primitives while designing for concurrency: a task-based system is enough for the job

Other patterns

There are a lot of other patterns used in concurrent design. [McCool12] provides a much more extensive collection of patterns to be used in the concurrent world. Patterns like *stencil*, *geometric decomposition*, *pack*, *expand*, *gather*, *scatter* are some of the patterns described in the book that we haven't touch at all here.

Although, not expressed in terms of tasks, [Buschmann07] also provides a series of patterns that can be used to build concurrency. Some of the patterns there encourage people to use synchronization primitives, but as previously shown [Teodorescu20b], one can always model them with tasks.

I believe that, by now, the reader should have a good intuition on how serial programming patterns can be transformed into concurrent patterns; how expressing the problems in terms of tasks can ease the design of concurrent applications.

Final words

This article provides a short catalogue of design patterns that are applicable for building concurrent applications. It can serve both as a quick introduction into designing concurrent applications for those who never designed concurrent applications with tasks, and also a refresher for those who are experienced with task-based programming.

One important point that the article tries to make is that there is no need to use synchronization primitives while designing for concurrency. A task-based system is enough for the job. Moreover, the patterns exposed here try to highlight that, in some cases, designing with such primitives it's much easier than doing multithreading with explicit threads and synchronisation primitives.

From a design perspective, it's much easier to reason in terms of these patterns compared to reasoning systems built with synchronisation primitives. If the preconditions are met (tasks are independent, as required by the patterns), then one can fully reason about the implications of using such a pattern. There are no hidden dependencies with other parts of the system (as opposed to a lock-based system; see [Lee06]). One can say that the concurrency of such a pattern is fully encapsulated within the pattern itself. This is a huge step forward for designing concurrent applications. ■

References

[Alexander77] Alexander, Christopher. *A pattern language: towns, buildings, construction*. Oxford university press, 1977

[Buschmann07] Frank Buschmann Kevlin Henney, Douglas C. Schmidt, *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing* (Volume 4). Wiley, 2007.

[concore] Lucian Radu Teodorescu, *Concore library*, https://github.com/lucteo/concore

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994

[Hoare85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall; http://www.usingcsp.com/cspbook.pdf

[Lee06] Edward A. Lee, *The Problem with Threads*, Technical Report UCB/EECS-2006-1, 2006, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf

[McCool12] Michael McCool, Arch D. Robison, James Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, 2012

[Pike13] Rob Pike, 'Concurrency Is Not Parallelism', https://www.youtube.com/watch?v=cN_DpYBzKso

[Robison14] Arch Robison, *A Primer on Scheduling Fork-Join Parallelism with Work Stealing*, Technical Report N3872, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3872.pdf

[Teodorescu20a] Lucian Radu Teodorescu, 'Refocusing Amdahl's Law', *Overload* 157, June 2020 available at https://accu.org/journals/overload/28/157/teodorescu_2795/

[Teodorescu20b] Lucian Radu Teodorescu, 'The Global Lockdown of Locks', *Overload* 158, August 2020, available at https://accu.org/journals/overload/28/158/teodorescu/

[Wikipedia] Wikipedia, 'Continuation-Passing Style', https://en.wikipedia.org/wiki/Continuation-passing_style

C++ Modules: A Brief Tour

C++20's long awaited module system has arrived. Nathan Sidwell presents a tourist's guide.

One of the major C++ 20 features is a module system. This has been a long time in coming. The idea predates C++98; it is about time C++ caught up with other languages! In this article, I'll show 3 example programs, using progressively more advanced organization of code. There are a number of call-out boxes answering a few questions the main text might suggest. You can read those separately.

Let's start with a simple example showing some modular concepts. Listing 1 is a module interface file – this is the source file that provides importable entities to users of the module.

The name of the file containing that code can be anything, but let's put it in `hello.cc`. Listing 2 is a user of that module.

We can compile our program using a module-aware GCC[1] with:

```
> cd ex1
> g++ -fmodules-ts -std=c++20 -c hello.cc
> g++ -fmodules-ts -std=c++20 -c main.cc
> g++ -o main main.o hello.o
> ./main
Hello World!
```

You'll notice there are some differences to using header files:

- You compile the module interface, just as a regular source file.
- The module interface can contain non-inline function definitions.

```
// file: ex1/hello.cc
module;
// legacy includes go here - not part of this module
#include <iostream>
#include <string_view>
export module Hello;
// the module purview starts here
// provide a function to users by exporting it
export void SayHello
  (std::string_view const &name)
{
  std::cout << "Hello " << name << "!\n";
}
```

Listing 1

```
// file: ex1/main.cc
import Hello; // import the Hello module,
              // its exports become available
#include <string_view>
int main ()
{
  SayHello ("World");
}
```

Listing 2

1. GCC's main development trunk and released versions do not yet provide module support. See the 'Implementations' box for details.

The road to standardization

ISO Working Group 21 (WG21) is responsible for C++. It holds meetings 3 times a year, to discuss new features and resolve issues with existing features. These physical meetings are on hold now, and various subcommittees hold virtual ones.

In 2016 a Technical Specification (N4592) was published, which specified a modules system. As implementors (such as me) experimented with this, a number of changes or clarifications were made during its path to incorporation into C++20.

Because of the pervasive use of header files as the way of describing interfaces, a particular difficulty is solving what may be phrased as the 'how do we get there from here?' problem. That took up a significant fraction of design and implementation effort.

- You need to compile the interface before you compile sources that import it.

The interface is a regular source file. It just happens to create an additional artefact to the usual object file – a Compiled Module Interface (CMI). That CMI is read by importers of the module, and then code can refer to entities exported by the module. It is this dependency that forces the compilation ordering. In this particular case, the CMI contains information about the **`SayHello`** function's declaration, but not (necessarily) about its body. If **`SayHello`** was an inline function, the body would also (most likely) be present in the CMI.

Do we need a new source suffix?

Often other module tutorials use a new source file suffix for the module interface file. This is user choice. The compiler doesn't need a new suffix – it's all still C++. Adding a new suffix means teaching your entire toolchain about the new suffix, which was too fiddly for me, and I control the compiler and am completely at home in an emacs config file!

As described in the build-systems box, prescanners need to scan all your sources, not just interface files, they gain nothing from distinguished interface names. If you do want to distinguish your interfaces, for the same reasons it's useful to distinguish header files from source files, you could augment another part of the filename – a '`-I.cc`' ending maybe? As we'll see further down, there are variations on module interfaces – should they be distinguishable from each other? (With yet more suffixes?)

Part of the reason may be due to history. The modules-ts did not have a specific syntax to denote a module interface, as opposed to a module implementation. The compiler had to be told via command line switch. It was one of my first contributions to suggest in-file syntax should make it clear.

Nathan Sidwell is a long-time developer of GCC, having discovered that Open Source is more rewarding than proprietary software, compilers are more rewarding than hardware, and hardware is more rewarding than Physics. He can be contacted at nathan@acm.org.

A module can export names in any namespaces it chooses. The namespaces are common across all modules, and many modules can export names into the same namespace

What's a Compiled Module Interface (CMI) and how is it used?

I've described a module interface as producing a CMI. That's a common implementation technique, but the standard itself makes no mention of such things, nor does it require them (the standard says nothing about object files either, by the way). Different compilers have taken different approaches to the CMI. For instance, Clang's CMI represents the entire source file, and is another step in the compilation sequence, from whence the object file can be generated for instance. GCC generates the CMI as a separate artefact containing only the items required by importers. The CMI is a serialization of the compiler's internal representation, but a more mergeable form than the usual PreCompiled Header (PCH) mechanism.

Rather than distribute source code, could one distribute a CMI? Not really. The CMI contains target CPU-specific information in addition to being very compiler-specific. Besides, users of a module will probably need source code to aid debugging. As mentioned above, the CMI may not contain all the source code information, so an object file would be needed too.

Why is the CMI not general? C++ itself requires certain architectural features to be locked down in the front end. For instance, `sizeof(int)` – consider instantiating a template on that value, we have to know what it is to get the correct specialization. Other pieces of the C++ language are implementation-defined, and to be portable all implementations would need to have the same behaviour. Underlying ABI decisions make themselves visible in the C++ front end, as it may or may not need to create temporaries in passing and returning. Don't forget, different Standard Libraries are not binary compatible – you cannot link object files built against different library implementations.

Command line options also affect source. For instance `-std=c++20` will allow rather different code, and enable different standard library code to `-std=c++17`. If you disable exceptions with `-fno-exceptions`, you'll have differences in the internal representation streamed. The CMI data is probably tightly related to several command line options.

While CMIs might not be interchangeable, both GCC and Clang have extended the Itanium ABI so that their object files remain link-compatible.

- The CMI is a caching artefact, recreateable on demand.
- We already have a code distribution mechanism. It is source code.

You may notice that modules are not namespaces. A module can export names in any namespaces it chooses. The namespaces are common across all modules, and many modules can export names into the same namespace. An importer of a module has to use a qualified name to refer to a module's exports (or deploy using-directives).

You'll also have noticed that the main program had to `#include <string_view>`, even though the interface had already done so. The interface had done this in part of the file that precedes the module itself, and that part is not visible to importers. As the user code needs to create a `std::string_view`, it needs the header file itself. The header include and the import can be in any order. I'll get more into detail about this later, as it is an important bridge from today's code to the future's module code.

Export

You'll see the example used the resurrected `export` keyword in two places:

- `export module Hello;`
  `// declare the interface of a module`
- `export void SayHello (…);`
  `// make a declaration visible to importers`

The first use is a *module-declaration*, a new kind of declaration specifying the current source file is part of a module. You can only have at most one of them, and there are restrictions on what can appear before it. The intent is that you won't get surprised with it buried in the middle of a file. As you might guess, there's a variant of the *module-declaration*, which lacks the `export` keyword. I'll get to that later.

The second use allows you to make parts of the module interface visible to importers, and most importantly its lack allows you to keep parts of the interface private to the module. Only namespace-scope nameable declarations can be exported. You can't export (just) a member of a class, nor can you export a specific template specialization (specializations are not found by name). You cannot export things from within an anonymous namespace. You can only export things from the interface of a module (see Listing 3).

If you export something, you must export it upon its first declaration. This is like declaring something `static` – you have to do so on its first declaration, but a later redeclaration can omit the `static`. In fact, `export` is described in terms of linkage – it's how you get *external-linkage* from

```
export module example;
// You can export a class.
// Both it and its members are available (usual
// access restrictions apply)
export class Widget { … };
namespace Tool {
 // export a member of a namespace
 export void Frobber ();
}
// export a using declaration (the used things must
// be exported)
export using Tool::Frobber;
// export a typedef
export using W  = Widget;
// export a template definition. Users can
// instantiate it
export template<int I> int Number () { return I;}
// you cannot explicitly export a specialization,
// but you can create them for importers to use
template<> int Number<0> () {
  return -1; /* Evil! */ }
```

Listing 3

C++ already had **export** as a keyword ... but **module** and **import** are new. Will that cause problems?

Module ownership

Module ownership is a new concept. Declarations in the purview (after the *module-declaration*) of a module are owned by that module. No other module can declare the same entity. The module specification has been carefully designed to not *require* new linker technology. In general, module ownership can be added to the symbol-name of an entity, at the object-file level. You're probably familiar with overloaded functions having mangled names, so that the linker can distinguish between **`int Frob (int)`** and **`int Frob (double)`**. Module ownership can be implemented by extending that mangling, and that is just what the Itanium ABI does (used on Linux and many other systems).

However, there is a design trade-off. Should exported names be link-compatible with their non-modular equivalent? I.e. is it possible to create a header file with just the exported declarations of a module, and have that useable in module-unaware code? An alternative way of phrasing the question is whether modules exporting the 'same' entity should result in multiple definition errors (it is ill-formed). The Itanium ABI takes that approach, which is known as *weak* ownership.

The alternative *strong* ownership includes the module ownership in the exported symbols too, or uses new linker technology.

inside a module. Declarations with external linkage are nameable from other modules.

So, what happens if you omit the **`export`** inside a module? In that case, you get a new kind of linkage – *module-linkage*. Declarations with *module-linkage* are nameable only within the same module, as a module can consist of several source files, this is not like the *internal-linkage* you have with **`static`**. It does mean that two modules could both have their own **`int Frob (int)`** functions, without placing them into globally unique namespaces.

Types (including typedefs) can be exported (or not exported), in the same way as functions and variables. Types already have linkage (but typedefs do not). Usually we don't think about that, because we use header files to convey such information and they textually include the class or typedef definition. Modules has more rigorous formulation of linkage of these entities that do not themselves generate code (and hence object-level symbols).

You can also export imports (see Listing 4).

Here I've imported and re-exported **`<string_view>`**, (wait, what? importing a header file!? I'll get to that) so that users do not need to **`#include`** (or import) it themselves. To build this, you will need to process **`<string_view>`**:

```
> cd ex2
> g++ -fmodules-ts -std=c++20 -c \
  -x c++-system-header[2] string_view
> g++ -fmodules-ts -std=c++20 -c hello.cc
> g++ -fmodules-ts -std=c++20 -c main.cc
> g++ -o main main.o hello.o
> ./main
Hello World!
```

```
// file: ex2/hello.cc
module;
#include <iostream>
export module Hello;
export import <string_view>;
// importers get <string_view>

using namespace std; // not visible to importers
export void SayHello (string_view const &name)
{
 cout << "Hello " << name << "!\n";
}
// file: ex2/main.cc
// same contents as ex1/main.cc
```

Listing 4

World in transition

So, how do I write my lovely new modules, but have them depend on olde worlde header files? It'd be unfortunate if it could only use modules. Fortunately there's not one, but two ways to do this (with different trade-offs).

You saw the first way in the early example. We had a section of the source file before the module-declaration. That section is known as a Global Module Fragment (GMF). It's introduced by a plain **`module;`** sequence, which must be the first tokens of the file (after preprocessing and comment stripping). If there is such a GMF, there must be a *module-declaration* – you can't just have an introduced GMF, why would you need that? The contents of the GMF must entirely consist of preprocessing directives (or comments). You can have a **`#include`** there, but you can't have the contents of that **`#include`** directly in the top-level source. The aim of this design is to make scanning for the *module-declaration* simple. Both the introductory **`module;`** and the *module-declaration* must be in the top-level source, unobscured by macros.

New keywords

C++ already had **`export`** as a keyword, exported templates were removed in C++11, but the keyword remained reserved, but **`module`** and **`import`** are new. Will that cause problems? There is known code that uses **`module`** and **`import`** as identifiers in their external interfaces. It would cause difficulty if those suddenly became unusable.

The C++ committee took care to specify that the lexing and parsing of **`module`** and **`import`** declarations was context sensitive. Code using those tokens as identifiers will largely be unaffected, and if they are there are simple formatting workarounds.

2. As **`string_view`** has no suffix, you need to tell G++ what language it is. The c++-system-header language specifies (a) searching on the system **`#include`** path and (b) with **`-fmodules-ts`**, specifies building a header-unit. Other possibilities are **`c++-header`** (automatically recognized with a variety of typical header file suffixes) and **`c++-user-header`** (use using #include path).

Modules can get access to regular header files, and not reveal them to their users – we get encapsulation that is, in general, impossible with header files.

In this way, modules can get access to regular header files, and not reveal them to their users – we get encapsulation that is, in general, impossible with header files. Hurrah!

There is a missed opportunity with this kind of scheme. The compiler still has to tokenize and parse all those header files, and we might be blocking the compilation of source files that depend on this module. That's unfortunate. Another scheme to address this is *header-units*. Header units are header files that have been compiled in an implementation-specified mode to create their own CMIs. Unlike the *named-module* CMIs that we've met so far, all *header-unit* CMIs declare entities in the Global Module. You can import *header-units* with an import-declaration naming the header-file:

```
import <iostream>;
```

This import can be placed in the module's purview, without making it visible to importers.

Naturally, as *header-units* are built from header files, there are issues with duplicate declarations and definitions. But we can make use of the One Definition Rule, and extend it into this new domain. Thus *header-units* may multiply declare or define entities, and be importable into a single compilation. Unlike header files, importing a *header-unit* is not affected by macros already defined at the point of the import – the meaning of the *header-unit* is determined by the macros defined when it was compiled to a CMI.

Not all header files are convertible to *header-units*. The goal here is to allow most of them to be, generally the well-behaved header files. This work derives from Clang-modules, which was an effort to do this seamlessly without changing source code.

One thing *header-units* do, which named modules do not, is export macros. This was unfortunately unavoidable as so many header files expose parts of their interface in the form of macros. Named-modules never export macros, even from re-exported *header-units*.

Splitting a module

So far I've only shown a module consisting of a single interface file. You can split a module up in two different ways.

The simplest way is to provide *module-implementation* files, distinct from the interface. An implementation file just has a *module-declaration* lacking the `export` keyword (it doesn't export things). While a module must have only one interface file, it can have many implementation files (or none at all). The implementation files implicitly import the interface's CMI, but themselves only produce an object file. If you think about modules as glorified header files, then this is the natural separation of interface and implementation (but you're probably missing out).

The interface itself can be separated into *module-partitions*. Partitions have names containing exactly one `:`. These themselves can be interface or implementation partitions depending on whether their *module-declaration* has the `export` keyword or not. Interface partitions may export entities, just as the primary interface does. These interface partitions must be re-exported from the primary interface. The partitions may also be imported into any unit of the same module.

- Partitions provide a way to break a large interface into smaller chunks.
- Partitions are not importable into different modules. The partitions are invisible outside of their module.
- Implementation partitions provide a way to make certain definitions available inside the module only, but have users aware of the type (for instance).

For example we could break our original example up as shown in Listing 5, overleaf.

In the primary interface, the three imports can be in any order. That's one of the design goals – import order is unimportant. You can see that the import syntax for a partition doesn't name the module. That's also important, so that there is no temptation to import into a different module.

Implementations

I know of 4 popular compiler front ends that are on the path of implementing C++20 modules support.

- **The Edison Design Group FE**. This is a popular front end for many proprietary compilers. They implemented the original `export` specification and gained an awful amount of knowledge about pitfalls in combining translation units. I do not know the current state of the implementation, nor do I know implementation details.
- **The Microsoft Compiler**. This is complete, or very nearly so. The main architect of the modules-ts, Gabriel dos Reis, is at Microsoft, and has guided that design. I believe this is the most complete implementation.
- **Clang**. The Clang FE has provided an implicit module scheme for use with header files for some time. Much of that experience went into the header-unit design. Richard Smith of Google has guided that design.
- **GCC**. I have been working on an implementation for GCC. This is currently on a development branch, and not in a released version. Godbolt (https://godbolt.org) provides it as an available compiler 'x86-64 GCC (modules)'. The current status (along with build instructions and list of unimplemented features) is described at https://gcc.gnu.org/wiki/cxx-modules. I try not to regress its state, and I hope to merge it soon.[1]

The latter 3 implementations (at least), have a lazy loading optimization. Importing a module does nothing beyond annotating symbol tables noting that an import contains something with a particular name. It is only when user code mentions a name that the relevant parts of the import are read in. The same is true for the macros of header-units. Thus importing is even cheaper than #including than might be expected.

Apologies to any other C++ compilers that I have failed to mention.

1. Det er vanskeligt at spaa, især naar det gælder Fremtiden. [It is difficult to make predictions, especially about the future.] Probably a Dane other than Niels Bohr, https://quoteinvestigator.com/2013/10/20/no-predict/.

```
// file: ex3/hello-inp.cc
module;
#include <string_view>
// interface partition of Hello
export module Hello:inter;
export void SayHello
  (std::string_view const &name);

// file: ex3/hello-imp.cc
module;
#include <iostream>
// implementation partition of Hello
module Hello:impl;
import :inter; // import the interface partition
import <string_view>;

using namespace std;
void SayHello (string_view const &name)
// matches the interface partitions's exported
// declaration
{
  cout << "Hello " << name << "!\n";
}

// file: ex3/hello-i.cc
export module Hello;
// reexport the interface partition
export import :inter;
import :impl; // import the implementation
partition
// export the string header-unit
export import <string_view>;
// file: ex3/main.cc
// same contents as ex1/main.cc
```

Listing 5

Here are the build commands:

```
> cd ex3
> g++ -fmodules-ts -std=c++20 -c \
  -x c++-system-header string_view
> g++ -fmodules-ts -std=c++20 -c hello-inp.cc
> g++ -fmodules-ts -std=c++20 -c hello-imp.cc
> g++ -fmodules-ts -std=c++20 -c hello-i.cc
> g++ -fmodules-ts -std=c++20 -c main.cc
> ar -cr libhello.a hello-{i,inp,imp}.o
> g++ -o main main.o -L. -lhello
> ./main
Hello World!
```

Note that in this example there was no need to import the implementation partition – it had no semantic effect.

Module ABI stability

An important part of module interface design is control of the aspects that are visible to users. Generally, the parts of the interface that can result in the importer emitting code are part of the ABI of your module. You want to control that.

The One Definition Rule

The One Definition Rule specifies that in a complete program there can only be One Definition of certain types of entities. And for those that can have multiple definitions (class, inline function, template instantiations), it places restrictions specifying how all those definitions are equivalent. It is the source of many 'ill-formed, no diagnostic-required' clauses in the standard – you the poor user get to figure it out.

Modules, and header-units, make it much harder to have silent ODR violations, which is good. The down side is you shouldn't be surprised when it finds ODR violations in your existing code as you convert to modules. At least you'll get a diagnostic rather than a land mine.

How will build systems be affected?

C++ build systems will need to change. The hardest build is build-from-scatch, when one does not have dependency information from a previous build.

In a header-only world, their problem was much simpler – all sources files can be built in parallel. Unless of course there are generated headers, and usually build systems suck with those. But now, we have interdependencies between source files. We cannot build the importers of module **`Foo`**, until we've built module **`Foo`** – we have the equivalent of generated headers all over the place! To make the problem harder, there's no defined mapping between module names and the source file name of the interface.

There are essentially two approaches to solving this.

- **Prescanners**. A prescan stage processes all the source files. Fortunately the design is such that this scanning is relatively simple, if one's happy with over-estimating the dependencies. The module-declarations and import-declarations must appear on lines by themselves, without the **`module`**, **`export`** or import keywords being obscured by macros. They're pretty much like preprocessor directives without the leading **`#`**. If one ignores everything else – including **`#if`** lines, one will get the maximal set of dependencies. To further simplify, all the imports of a module must appear immediately after the module-declaration – you can't place one later in the middle of the module. For more accurate dependencies one would have to track **`#if`**s and macros. With this information computed, the build can launch compiles in the correct order, and inform each of the locations of the Compiled Module Interfaces (CMIs) it will require.
- **Dynamic build graph**. The compiler could consult an oracle whenever it meets an *import-declaration*, and inform the same oracle whenever it meets a *module-declaration* and produces a CMI. If the oracle is the build-system, it can modify its dependency graph, build the needed CMI(s) and then inform the compiler of the location. Because of the requirement of import placement, this can even be parallelized somewhat!

In both cases the determined build graph can be retained for a subsequent incremental build.

Note that in an unconstrained parallel build, a clean build of modular code is likely to be slower than that of #include builds – it's constrained by the module dependency tree. However, an incremental build is likely to be much faster, because header-files do not need to be reparsed all the time. Google's experience with Clang's implicit modules showed this to be a significant win.

Every exported inline function's body is visible to importers (they need to refer to the entities it names), and changing the body can change the ABI of a module. To that end, one significant change has been made to in-class function definitions. They are no-longer implicitly inline in a module's purview! The implicit functions are still inline, as are lambdas. This means you no longer have to separate the definitions of your non-inline member functions (including template definitions), from their in-class declaration.

Onwards!

I hope the examples here have shown you a flavour of what is available with modules. I kept the examples simple, to show some of the core module concepts, particularly how non-modular and modular code can interact.

As mentioned elsewhere, I believe the Microsoft implementation is the most advanced, and has been used for production code. Of the other implementations, GCC's is more complete than Clang's (mid 2020).

Unfortunately, for GCC one must use Godbolt, which is awkward for the more advanced use, or build one's own compiler, which is a steep cliff to climb for for most users. To make things even more exciting, those that have played with GCC have fallen over bugs. As with any major new feature, ensuring it is correct is difficult, and users have imaginative ways of exercising things. Don't let that put you off though, user bug reports are helpful. ■

The Edge of C++

Everything has limits. Deák Ferenc explores the bounds of various C++ constructs.

Everything we interact with in our daily lives (well, except the universe) has a boundary that constrains its existence to within well-defined limits. There are borders delimiting a country, there are walls keeping out bluer than white walkers, there is a finite number of bits that a variable can reach and, of course, there is the maximum quantity of source a C++ compiler can swallow without choking.

In our daily usage of C++, we rarely reach these limits, but regardless, the almighty Standard covers these fringe situations too. This article, in which we will explore the outer edges of some of the most common compilers, is based on 'Annex B – (informative) Implementation quantities' [ANNEX-B] of the (current) C++ Standard.

In this article, I will walk you through these limitations and what they mean for your daily life. I will present a tool for generating small test source files for testing each of the specific limits from Annex B and that will push the compilers to *their* limits.

Annex B

No, I am not going to include Annex B in the article. It would be a waste of paper and we are trying to be as environmentally conscious as possible, so anyone interested can fetch it (from [ANNEX-B]), and I will just give a short overview of what it is.

In the C++ Standard, Annex B lists the maximum recommended values for various code snippets from a C++ application that the Standard writers recommend that a compiler should support. From the Standard:

> However, these quantities are only guidelines and do not determine compliance.

For example, it is recommended that the supported number of arguments in one function call should be at least 256. Certainly, this sounds like a pretty big number and no-one would be expected to type in 256 arguments by hand, but consider that today a lot of the code that is being compiled is first generated by code generators (think about Google's protobuf compiler for example, or just the unreadable output of a software modeling/CASE application, Qt's resource compiler or any other applications out there which generate code for you). You may find yourself in a situation where code being generated is heading towards this limit.

The actual limits imposed by the compiler

All the current compilers I have tested have a page ([GCC], [CLANG], [MSVC]) where they present the limitations actually imposed by their implementations. However, not all the available limits presented in Annex B were to be found in all the documented limitations, and not all the compilers have identical values.

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

The test suite

As mentioned before, the main purpose of this article is to provide a set of tests for compilers to test the supported edge situations. The code is generated by an application, for fun's sake written in the **`go`** programming language, and it is available at the [GITHUB] location. Everyone is free to download it, modify it and extend it to fit their needs.

The test suite is contained in a big **`json`** package where each entry is of the format shown in Listing 1, where most of the fields are self-explanatory; however, **`testName`** must be mapped to one of the functions in the **`go`** program, which parses this **`json`**, and calls the specific methods, for each value in the **`count`** field.

As a side note, some of these test cases were intended to test a specific feature of the compilers, but unwittingly highlighted an error somewhere else in the product. I took the decision to leave them as they are because highlighting these errors might be useful for compiler writers on the quest to continuously improve their products.

The compilers

All the tests I have performed on a computer using dual boot between two operating systems: Firstly a brand new shiny Ubuntu 20.04 just downloaded from Canonical which by default comes with the following compilers:

- **`g++ 9.3.0`** (installed via **`apt`**)
- **`clang 10.0.0`** (installed again via **`apt`**)
- **`icc (ICC) 19.1.2.254 20200623`** installed as a by-product from a trial version of Intel Parallels Studio

And secondly, under Windows 10:

- **`msvc`** from Visual Studio 2019 (Version 16.4.5)

I deliberately chose not to use a locally compiled version of any of those compilers. I tend to stick to the mainstream Linux distributions, and use what is available for the largest communities of programmers right out of the box, so making a highly personalized compiler would not have been

```
{
  "run": true,
  "testName" :
    "parameterCountInFunctionDefinition",
    "count": ["256"],
    "minimum": "256",
    "description":
      "Parameters in one function definition
      ([dcl.fct.def.general]"
}
```

Listing 1

everything we interact with in our daily lives has a boundary that constrains its existence to within well-defined limits

an ideal comparison ground for everyone who uses default compilers on their OS.

Some of these test cases required the activation of C++17 features; however, I consider that in 2020 this should not be such a big issue.

Timing issues

In the test results, I intentionally did not include a precise measurement of the time it took each test to compile. That would have only made sense for my computer, and if someone repeats the test on a much slower or faster computer, the results they obtain will be significantly different.

Where I have observed a noticeable difference between the various compilers, I have added my comments regarding the specific case.

The compilers' own test suites

Before digging deeper into the subject, I have to mention that both **`gcc`** and **`clang`** come with exhaustive test suites meant to verify the correct functionality and compliance (with the Standard) of the compilers. However, I did not find a dedicated test suite for the edge situations I am researching through this article, so I thought that providing a unified set of tests for all the C++ compilers would be beneficial.

Unfortunately, I did not find any test suite for the Microsoft compiler nor for Intel's compiler, considering the closed source nature of the product, but I would love to hear from developers who actually work(ed) on Microsoft's C++ compiler to see whether they have considered these test cases too.

Numbers

For the test cases, I intentionally used numbers that are powers of two. Only for very special cases did I dig deeper and identify a number outside of this family. For most of the test cases, I specifically tested against the Standard-recommended values, and where for some test cases I have pushed the compilers a bit further, there is a note in the test case.

The tests

Most of the tests are represented as a single generated C++ file; however, some cases required that some of the tests are joined together. For example, testing the maximum number of arguments really makes sense with the maximum number of parameters a function can have.

These small applications were carefully engineered to cover all the required edge cases and are all compilable independently of each other. Setting **`generateMakefile`** to **`true`** in the json will generate a Makefile when the test generator is run beside the CPP files.

Furthermore, you can request measurements of execution time (and other important data) using the **`timeCompilation`** flag, and the **`timeFlag`**s in the json. I currently use **`"-f '%E,%M'"`** to measure the time taken in seconds (**`%E`**) and the amount of memory used (**`%M`**) by the process.

So as not to depend on data from only one invocation of the compiler, you can gather an average execution time for the compiler compiling the same source by specifying the **`"compilationTimes"`** property to be the number of compiler invocations you want.

```
int main() {
  for (int f0 = 1; f0<256; f0++ )
    if (f0 % 2 == 0)
      for (int f1 = f0; f1<256; f1++ )
        if (f1 % 3 == 0)
          for (int f2 = f1; f2<256; f2++ )
            if (f2 % 4 == 0)
              ...
```

Listing 2

There are places in the tests where local (global) variables are initialized. For ease and in order to get a consistent and reproducible behaviour between test runs, all of them are initialized to 1. I found no difference in the compilers' performance whether I used a set of random numbers or just used 1.

All of the tests require the output of some values to the screen, so I used the standard **`iostream`** header with **`std::cout`** to print out all necessary values.

Nesting level of iteration, selection, compounds statements – nestingOfStatements

For this test, I generated a simple source file containing alternating **`for`** and **`if`** statements, like the sequence in Listing 2.

This seemed to be complex enough to prevent the compiler from optimizing out everything while still generating assembly code that was not itself overly complex.

Regardless, no-one in this life should be required to handle applications in which the nesting level reaches even half of the Standard-recommended maximum value to support, which is 256. (Maybe this is why **`clang`**, being a pragmatic compiler, actually got stuck at 128 and was killed after five hours of struggling with the generated source consisting of 256 nested statements.) **`gcc`**, though, had no problem compiling applications which contained up to 1024 nested levels. More, I did not dare try.

`icc` had no problems generating code for up to 256 nested levels, but **`msvc`** does not support the depth of 256. It actually gives an error at 166:

```
nestingOfStatements-166.cpp(169): fatal error
C1061: compiler limit: blocks nested too deeply
```

but compiles fine for 164 levels.

gcc	clang	msvc	intel
1024	128	164	256

Nesting levels of conditional inclusion – nestingLevelOfConditionalInclusion

This test required the definition of a specific number of identifiers, all of which could be used as tests in a conditional check. If all of them evaluated

optimizers in today's compilers ... do the calculations themselves and substitute the results in the generated code

```
#include <iostream>

constexpr int z() {
  return 0;
}
int main() {
  volatile int i = 0;
  volatile int *volatile p1=&i;
  volatile int *volatile *p2 = &p1;

  * & z()[* & z()[* & z()[&p2] ] ] = 4;
  std::cout << i << std::endl;
}
```

Listing 3

to true, the proper header file for writing out the actual number for this test was included. The code generated was like:

```
#define COND_0 1
#define COND_1 1
...
#if defined COND_0
 #if defined COND_1
  ...
   #include <iostream>
  ...
 #endif
#endif
```

No compilers had any issue compiling the code up to 512 identifiers, which is double the Standard-recommended value to support; however, **clang** was much slower than **gcc**.

gcc	clang	msvc	intel
512	512	512	512

Pointer, array, and function declarators modifying something – pointerAndArrayDeclaratorsModifyingSomething

I have to admit, this was one of the trickiest cases I had to generate code for, as the optimizers in today's compilers are simply too clever. They instantly see through your intentions and, throwing out all your efforts to generate code for calculating values, instead do the calculations themselves and substitute the results in the generated code. I really had to use a lot of trickery.

For example, Listing 3 (overleaf) is the code generated for 4.

As expected, it prints out 4. Some explanations: firstly, if there is no **volatile**, the compiler simply ignores all the code and just generates the required assignment. The weird looking expression of `* & z()[* & z()[* & z()[&p2] ] ] = 4;` is actually equivalent to `**&p2[0] = 4;` but I wanted to use both pointer arithmetic, array indexing and function in the same expression, thus ended up with this monstrosity.

gcc had no problems compiling up to 1024 modifiers, **clang** complained at a certain point that **fatal error: bracket nesting level exceeded maximum of 256**; however, if I specified **-fbracket-depth=1024**, it compiled without any issues.

msvc and **icc** again had no problems compiling up to 1024.

gcc	clang	msvc	intel
1024	1024	1024	1024

Nesting levels of parenthesized expressions – nestingLevelsOfParenthesizedExpressionsInAFullExpression

Because the compiler can be very effective at optimizing code by pre-calculating values in the compilation phase, this test generated a complex parenthesized expression to calculate the summation and multiplication of various numbers. **gcc** had no issues with the depth of the expression up to 1024 (four times the Standard-recommended number); however, **clang** gave a very clear error message in the form of **fatal error: bracket nesting level exceeded maximum of 256** and I also appreciated the suggestion on the next line on how to fix it: **use -fbracket-depth=N to increase maximum nesting level**. After using this parameter, **clang** compiled without problems.

msvc and **icc** had no problems compiling nesting parentheses up to 1024, which is a pretty large value for this purpose, so I concluded that this was an acceptable value for this test case as some compilers (well, all except **gcc**) started showing error messages for 2048.

gcc	clang	msvc	intel
2048	1024	2048	2048

Number of characters in an internal identifier or macro name – identifierOrMacroNameLength

This was an easy run: just define a macro with a long random name, then a function with a different long random name containing a variable with a third long random name being assigned to the macro. Then, call this function. Most of the tested compilers had errors when compiling code with variable names as long as 8192 characters, except **msvc** which conjured up the message:

```
identifierOrMacroNameLength-8192.cpp(3): fatal
error C1064: compiler limit: token overflowed
internal buffer
```

msvc proved to be successful for 2048.

gcc	clang	msvc	intel
8192	8192	2048	8192

These test cases were **specifically engineered** for a unique purpose, and they are **not real life** situations

Number of characters in an external identifier – externIdentifierNameLength

Almost as easy as the previous test, I just had to use a small trick. To avoid multiple compilation units for the extern variable, I defined it just after `main` as follows:

```
#include <iostream>

int main() {
  extern int vxvl;
  std::cout << vxvl << std::endl;
}
int vxvl = 4;
```

Most of the tested compilers had no issues in compiling code with variable names up to 8192 characters, which I consider to be more than enough, except `msvc` which gave up with a similar error message to the previous case, but it succeeded for 2048.

gcc	clang	msvc	intel
8192	8192	2048	8192

External identifiers in one translation unit – externIdentifiersInOneTranslationUnit

The code generated for this case pretty much follows the recipe for the previous case, just varying the number of identifiers. Here, to my surprise, `clang` crashed during something that – in the printed stack trace – looked like a recursive call when it tried to compile the Standard-suggested value of 65536. `gcc` also had its fair share of struggles with this value, taking several minutes; however, it completed its task successfully. `icc` gave up with the following error:

```
externIdentifiersInOneTranslationUnit-
65536.cpp(65540): internal error: bad pointer
```

so I had to lower my expectations. `clang` successfully managed to compile 8192 external identifiers, and `icc` managed 4096.

`msvc` really had no problems compiling the test case with 65536 values.

gcc	clang	msvc	intel
65536	8192	65536	4096

Identifiers with block scope declared in one block – identifiersWithBlockScopeDeclaredInOneBlock

This test just consisted of generating a long list of variables in a block and seeing when the compiler complained, but all the tested compilers successfully compiled up to 8192 local variables.

gcc	clang	msvc	intel
8192	8192	8192	8192

```
#include <iostream>

int main() {
  int arr[] = {1, 1, 1, 1};
  auto volatile [v0, v1, v2, v3] = arr;
  int i = v0 + v1 + v2 + v3;
  std::cout << i << std::endl;
}
```

Listing 4

Parameters in one function definition – parameterCountInFunctionDefinition

This is one of the test cases which was joined together with another, namely 'Arguments in one function call', because it just made sense. The application generates a function with the required number of parameters, generates a list of variables of different type, and calls the function with the required number. None of the tested compilers had issues compiling functions with up to 4096 parameters, which is 16 times the recommended amount, so I consider that to be a fair number.

gcc	clang	msvc	intel
4096	4096	4096	4096

Structured bindings introduced in one declaration – structuredBindingsInOneDeclaration

The code generated for this is a larger scale of the one in Listing 4.

Some of the tested compilers (well, all except `icc`) had no issues compiling code with up to 8192 values in the structured binding expression. To my huge surprise, however, this is one of the tests `gcc` proved to be slower at than `clang`, but both compiled the test files nicely.

My other surprise came from `icc`, which gave a core dump upon compiling 8192 (see Listing 5) but in the end, 4096 seemed like a good number for `icc`.

gcc	clang	msvc	intel
8192	8192	8192	4096

Macro identifiers simultaneously defined in one translation unit – macroCountInOneTranslationUnit

This was one of the easiest tests to come up with: just generate a file with enough macros and let the compilers go wild on them. `gcc` and `icc` had no problems sorting out files (at blazing speeds) containing up to 65536 macros; however, `clang` started choking after 8192 with a coredump. A similar fate awaited `msvc`:

```
macroCountInOneTranslationUnit-8192.cpp(8197):
fatal error C1009: compiler limit: macros nested
too deeply`
```

Each of the tested compilers shines in some areas and performs poorly in others

so I had to lower my expectations and the number of generated macros to 256.

When even this number gave a compiler error (not a compile error), I started thinking that maybe my test case is simply wrong, maybe I expect too much from the macro engine of **msvc**, or that the test with the following logic is simply not good:

```
#include <iostream>

#define V0 1
#define V1 V0 + 1
#define V2 V1 + 1
#define V3 V2 + 1
#define V4 V3 + 1

int main() {
  std::cout << V4<< std::endl;
}
```

From the error message, I felt that somehow this specific test case must have stepped on the toes of the **msvc** compiler, so I concluded that the test is using the wrong approach for this situation because I felt that no (decent) compiler would have problems with 256 macros defined in a source file so the problem must be the recursive substitution part of it. However, since it managed to annoy two of the compilers to the point of breaking, I decided to leave it in here; maybe someone will have a look at these cases in one of the development teams.

gcc	clang	msvc	intel
65536	8192	128	65536

Parameters in one macro definition – parametersInMacroDefinition

This test was very simple to construct, involving a macro, similar to the **parameterCountInFunctionDefinition**. This test case was implemented together with the 'Arguments in one macro invocation' test, since it sort of made sense to have both run together.

All the compilers did very well with the code with up to 4096 parameters (except the Microsoft compiler, see below) which is several time above the supported number recommended by the Standard. On [GCC], it is mentioned that **gcc** allows up to **USHRT_MAX** number of arguments, which should be at least 65535. 65535 worked nicely, but the gremlin woke up somewhere inside and I had to try to run with 65536.

gcc provided a cute (but weird) error:

```
parametersInMacroDefinition-65536.cpp:5: error:
macro "M" passed 65536 arguments, but takes just 0
    5 |   int v = M(1, 1, 1,
```

Seemingly there was an overflow somewhere deep inside **gcc**. **clang** threw a tantrum in form of a coredump for the same number. **icc** compiled without any complains.

Microsoft's own compiler was very consistent with the amendments mentioned in [MSVC]. It accurately gave a warning that 127 is the maximum number of parameters supported for these situations.

clang successfully compiled for 9216 parameters but failed for 10240, so I decided that the maximum supported value must be somewhere between the two.

gcc	clang	msvc	intel
65535	9216	127	65536

Characters in one logical source line – charactersInOneLogicalSourceLine

This test case was just about generating a long list of summations, that in the end will print out the number of characters in the test case. The following listing, for example, gives the source for 15.

```
#include <iostream>
int main() {
int a=9+2+2+2 ;
  std::cout << a << std::endl;
}
```

gcc struggled with the Standard-recommended value (65536), but after a while it completed the operation successfully. **clang**, to my big surprise,

```
structuredBindingsInOneDeclaration-8192
": internal error: ** The compiler has encountered an unexpected problem.
** Segmentation violation signal raised. **
Access violation or stack overflow. Please contact Intel Support for assistance.

icc: error #10105: /home/fld/intel/compilers_and_libraries_2020.2.254/linux/bin/intel64/mcpcom: core
dumped
icc: warning #10102: unknown signal(0)
icc: error #10106: Fatal error in /home/fld/intel/compilers_and_libraries_2020.2.254/linux/bin/intel64/
mcpcom, terminated by unknown
compilation aborted for structuredBindingsInOneDeclaration-8192.cpp (code 1)
```

Listing 5

crashed again; however, I am not sure whether it was due to the very long sequence of operations handled in a peculiar mode by **clang** or due to the line length. Since I personally don't consider this to be the most important test case, I just let it lie. This test case will not work correctly for values under 10, but lines with length under 10 should not be a struggle for any compiler.

msvc and **icc** had no problems compiling lines with the required length.

gcc	clang	msvc	intel
65536	16384	65536	65536

Characters in a string literal after concatenation – charactersInAStringLiteral

This is again was one of the easiest test cases: just generate a string long enough and run a **strlen** on it. This case might be useful for tools which are generating source code for embedding resources into C++ applications (such as aforementioned Qt's resource compiler). None of the compilers (except **msvc**, with a well defined limit from [MSVC]) I tested had any problems running with strings long as 131072 characters, double the Standard-recommended value.

gcc	clang	msvc	intel
131072	131072	65535	131072

Size of an object – sizeOfAnObject

This required some tricks in order to beat the optimizer, leading to the source in Listing 6 for 262144 (which is, by the way, the value recommended by the Standard) being generated.

It actually surprised me how far the optimizer can go in order to save memory, time and space for you. Unless you place some complex calculations and constraints on the values it has to manage, it will simply precalculate all the values for you without leaving a trace of their origins in the generated binary. Of course, I am talking about release builds with optimization turned on.

No compilers had problems in generating code (and running the generated executable) for class sizes up to 2097152, which is 8 times the Standard-required supported size.

gcc	clang	msvc	intel
2097152	2097152	2097152	2097152

Nesting levels for #include – filesnestingLevelsForIncludes

For this test, I created the required number of header files and placed them in the `inc` directory, with each header including the next one. The current iteration of the Standard suggested a supported nesting level of 256 here, and this is mentioned on [GCC] (but with a value smaller, specifically 200). Both **gcc** and **clang** subscribe to this 200, and we get a very specific error in the form of **error: #include nested too deeply**.

```
#include <iostream>
#include <numeric>

class A {
public:
  A() {
    std::iota(std::begin(c), std::end(c), 0);
  }
  void printer() {
    for(auto i=0ULL; i<sizeof(c); i++) {
      if(c[i] * 256 == i && i > 0) {
        std::cout << i ;
      }
    }
    volatile auto x = sizeof(*this);
    std::cout << x << std::endl;
  }
private:
  unsigned char c[262144];
};
int main() {
  static A a;
  a.printer();
}
```

Listing 6

icc and **msvc**, on the other hand, managed up to 256, which is considered a success.

gcc	clang	msvc	intel
200	200	256	256

Case labels for a switch statement – caseLabelsForSwitch

For this test, I implemented a simple random generator, which could pick values between 1 and the required value, and – in a long switch statement – printed out the square of that number. No compiler had problems compiling the code up to 16384, the value recommended by the Standard.

gcc	clang	msvc	intel
16384	16384	16384	16384

Non-static data members in a single class – nonStaticDataMembersOfClass

This was also one of the more wood-cutting types of work: just generate a class, with the required number of data members (and for simplicity's sake, all in one class) and sum those up. **msvc**, **gcc** and **clang** had no problems generating code for classes which contained 65536 data members, which is more than the double the recommended amount.

icc choked on that value (and for 32768, 16384, 8192 and 4096 too), but nicely compiled for 2048, which I found a bit strange (see Listing 7) because an application of the form in Listing 8 (generated for 4) does not

```
nonStaticDataMembersOfClass-4096
": internal error: ** The compiler has encountered an unexpected problem.
** Segmentation violation signal raised. **
Access violation or stack overflow. Please contact Intel Support for assistance.

icc: error #10105: /home/fld/intel/compilers_and_libraries_2020.2.254/linux/bin/intel64/mcpcom: core
dumped
icc: warning #10102: unknown signal(0)
icc: error #10106: Fatal error in /home/fld/intel/compilers_and_libraries_2020.2.254/linux/bin/intel64/
mcpcom, terminated by unknown
compilation aborted for nonStaticDataMembersOfClass-4096.cpp (code 1)
Command exited with non-zero status 1
```

Listing 7

possess a huge level of complexity, so theoretically should not be a huge problem for a compiler.

gcc	clang	msvc	intel
65536	65536	65536	2048

Lambda-captures in one lambda-expression – lambdaCapturesInOneLambdaExpression

Again, one of the easiest test cases: just generate the required number of variables, and a lambda trying to capture them. **msvc**, **gcc** and **clang** had no issues compiling lambdas capturing 8192 values, which I considered enough for even the most evil code generated by any code generator.

icc core-dumped for that value, but successfully compiled code generated for 4096 variables.

gcc	clang	msvc	intel
8192	8192	8192	4096

Enumeration constants in a single enumeration – enumerationConstantsInEnum

This was again one of the easiest cases: just generate an **enum** with enough members and let the compiler pick out a random value from them. No compiler had issues compiling code generated with up to 8192 values, which is the double of the indicated number to be supported in the Standard.

gcc	clang	msvc	intel
8192	8192	8192	8192

Levels of nested class definitions – nestingOfClasses

Nested classes used in projects when encapsulating information should provide a better overview of what the class is about, and what information to keep apart. However, too deep a nesting of inner classes will (after a while) produce unreadable code (personal opinion) and will possibly lead to a maintenance nightmare. This may be why Microsoft reduced the nesting level to a humanly manageable number (16) while other compilers keep their value at 256, the value recommended by the Standard.

gcc	clang	msvc	intel
256	256	16	256

Functions registered by atexit() – functionsRegisteredByatexit

The Standard recommends 32 here, but no compiler had problems generating code (which worked as expected) for sane values, although this was all actually dependent on my OS. On systems conforming to **POSIX**, the correct method for finding out the number of functions that can be registered for **atexit** is using the **sysconf** function with **_SC_ATEXIT_MAX** as a parameter.

```
#include <iostream>
class TestClass {
public:
  short int m_member0 = 1;
  unsigned short int m_member1 = 1;
  unsigned int m_member2 = 1;
  int m_member3 = 1;
};
int main() {
  TestClass tc; int v = 0;v += tc.m_member0;
  v += tc.m_member1;
  v += tc.m_member2;
  v += tc.m_member3;
  std::cout << v << std::endl;
}
```

Listing 8

The Windows SDK had a remark in the form that the number of functions that can be registered is limited by the available heap space.

gcc	clang	msvc	intel
64	64	64	64

Functions registered by at_quick_exit() – functionsRegisteredByat_quick_exit

According to the documentation, the difference between **std::exit** and **std::quick_exit** is the amount of cleanup done when the application exits (for example, calling static objects' destructors, or other fine nuances). The Standard recommends at least 32 functions, but I have found that registering 8192 is also alright with **gcc**, **icc** and **clang**. And because, sadly, this feature is among the ones for which there is no **POSIX**-assigned retrieval count, as is case for **atexit**, I concluded that 64, the same value as for **atexit**, should be a good value for this situation.

gcc	clang	msvc	intel
64	64	64	64

Direct and indirect base classes for a class – directAndIndirectBaseClassesOfClass

Making this test would have been much more easier if I had opted just to generate a bunch of classes as I did for **directBaseClassesOfClass**. However, what I did was to create a full binary tree with a number of nodes as close as possible to that required and generate a class hierarchy from this tree. A binary tree with 13 levels already contains a huge number of nodes and this pretty much covers the classes for the main part of our test. In cases where the requested number was not exactly one less than a power of two, I generated a set of additional classes that were added to the inheritance list for the tests' target **Derived** class, bumping the number of classes up to that required.

No compiler had any problems compiling code with values up to 65535.

gcc	clang	msvc	intel
65535	65535	65535	65535

Direct base classes for a single class – directBaseClassesOfClass

This test was also a straightforward one: I just had to generate a long list of base classes and a derived one from them. To prevent the compiler optimizing away the classes, for each class I stored a global static value in a class member (and also printed it out in the constructor), which incremented with every constructor call, and I also summed the values at the end.

No compilers had problems compiling code with up to 4096 generated direct classes, which is 4 times more than the number recommended by the Standard.

gcc	clang	msvc	intel
4096	4096	4096	4096

Class members declared in a single member-specification – classMembersDeclaredInASingleMemberSpecification

The code generated for a value of 5 is something like:

```
#include <iostream>
class A {
public:
  int v1 = 1, v2 = v1 + 1, v3 = v2 + 1,
    v4 = v3 + 1, v5 = v4 + 1;
};
int main() {
  A a;
  std::cout << a.v5 << std::endl;
}
```

```
  staticDataMemberOfClass-16384
  ": internal error: ** The compiler has
   encountered an unexpected problem.
  ** Segmentation violation signal raised. **
  Access violation or stack overflow. Please
  contact Intel Support for assistance.

  icc: error #10105: /home/fld/intel/
  compilers_and_libraries_2020.2.254/linux/bin/
  intel64/mcpcom: core dumped
  icc: warning #10102: unknown signal(0)
icc: error #10106: Fatal error in /home/fld/intel/compilers_and_libraries_2020.2.254/linux/bin/intel64/
mcpcom, terminated by unknown
compilation aborted for staticDataMemberOfClass-16384.cpp (code 1)
Command exited with non-zero status 1
```

Listing 9

so for the Standard-recommended 4096, the same logic is used. No compiler had problems compiling code for up to 16384 class members, which I considered enough for this purpose as I strongly advocate the principles of clean code, and recommend everyone to have at most one, or (in the worst case) a small group of members that logically belong together (and of course, don't forget to add comments to explain their purpose).

gcc	clang	msvc	intel
16384	16384	16384	19384

Final overriding virtual functions in a class – finalOverridingVirtualFunctions

This test case also requires a class hierarchy just as for `directAndIndirectBaseClassesOfClass` but also introduced virtual functions, to make the generation more fun. Since, even for small numbers, the code tends to be long and repetitive, I will not put any example code here, but feel free to check out the code generated for this case by the test application. The Standard recommends 16384 as the magic limit for this case, and I think that is indeed a very good number.

`gcc` and `clang` had no problems generating code for values up to 32768; however, `msvc` didn't manage to compile code generated for this value (neither the 32 bit compiler, nor the 64 bit one). Both failed with the error:

```
finalOverridingVirtualFunctions-32768.cpp(262149):
fatal error C1060: compiler is out of heap space`
```

I found this a bit strange, since neither of those compilers consumed too much memory while running. The real surprise came when I tried to compile with 16384 functions, and I was greeted by an internal error from the compiler:

```
finalOverridingVirtualFunctions
-16384.cpp(196618): fatal error C1001: An
internal error has occurred in the compiler.
(compiler file 'msc1.cpp', line 1528)
```

Finally, 8192 gave a result for `msvc` in the form of a compiled executable. Intel's `icc` could not even manage 8192, its final value is 4096.

gcc	clang	msvc	intel
32768	32768	8192	4096

Direct and indirect virtual bases of a class – directAndIndirectVirtualBaseClassesOfClass

This test case is also very similar to the one for `directAndIndirectBaseClassesOfClass` except that the inheritance must be virtual. The same language mechanisms were used to generate the required number of base classes, and the result is also very similar. Compiling with the limit set at 65535 took forever, but successfully completed both for `gcc` and `clang`. I did not dare try with a larger value. Interestingly, the code generated by `clang` is only 62 megabytes, while the one generated by `gcc` is 72 MB.

Sadly, after half an hour of struggle `msvc` gave up with the following error message:

```
directAndIndirectVirtualBaseClassesOfClass-
65535.cpp(65527): fatal error C1060: compiler is
out of heap space
```

and the same result was produced for 32768, and 16384 and 8192 too, so I came to the conclusion that the C++ compiler of Visual Studio 2019 can't handle these large applications, and I therefore reduced the maximum supported number to 4096.

`icc` really struggled with 4096. It took more than 30 minutes to compile the source file, so again, I have decided that this should be enough for it, and the same value applies for `msvc` too.

gcc	clang	msvc	intel
65535	65535	4096	4096

Static data members of a class – staticDataMemberOfClass

This test case consisted of generating a class with the specified number of public static members, of various numeric types. Following that is code to initialize these values to `1` and in the `main` function is code generated to sum up all the members.

None of the tested compilers had any issues with generated code containing up to 16384 static members except `icc`, which core-dumped again (see Listing 9).

gcc	clang	msvc	intel
16384	16384	16384	2048

Friend declarations in a class – friendsOfAClass

Friends of a class provide a useful back door into the internals of a class, but too many back doors aren't a very good approach to optimal application design, so you should not over-abuse them. The Standard indicates a value of 4096 and the compilers had no problems compiling with values up to 8192.

For this test, I generated a class and a combination of friend classes and functions, and then counted the number of private members via friend functions and classes.

gcc	clang	msvc	intel
8192	8192	8192	8192

Access control declarations in a class – accessControlDeclarationsInClass

I interpreted this test case as alternating `protected`, `public`, `private` of various data members, so the test generated is also nothing else but a long list of data members with alternating visibility, a set of `public` getter functions (the test application will print out the required number - 1, due to this last set being public) for the private and protected members, and a

```
: internal error: ** The compiler has encountered an unexpected problem.
** Segmentation violation signal raised. **
Access violation or stack overflow. Please contact Intel Support for assistance.

icc: error #10105: /home/fld/intel/compilers_and_libraries_2020.2.254/linux/bin/intel64/mcpcom: core
dumped
icc: warning #10102: unknown signal(0)
icc: error #10106: Fatal error in /home/fld/intel/compilers_and_libraries_2020.2.254/linux/bin/intel64/
mcpcom, terminated by unknown
compilation aborted for accessControlDeclarationsInClass-8192.cpp (code 1)
```

Listing 10

`main` function which simply generates a summation of all the members (which were set to one).

Some of the compilers I tested had no problems in generating code for alternating the visibility of data members up to 16384. To my surprise, this was one of the test cases where `clang` outperformed `gcc` in terms of speed.

`icc` sadly gave up at 16384 with the following error message:

```
accessControlDeclarationsInClass-16384.cpp(43698):
internal error: bad pointer
```

and threw an exception for 8192 (shown in Listing 10) but compiled nicely for 4096.

gcc	clang	msvc	intel
16384	16384	16384	4096

Member initializers in a constructor definition – memberInitializersInAConstructorDefinition

A not overly complicated test case: I just generated the required number of members in a class, generate code for the constructor and printed their sum in order to have some confirmation. Except for `icc`, no compiler had problems in compiling code generated for values up to 16384, and again, this is one of the test cases where `clang` was faster than `gcc`. However, for this high value `icc` came up with the following error:

```
memberInitializersInAConstructorDefinition-
16384.cpp(16720): internal error: bad pointer
```

Finally, `icc` settled at the value of 4096 (not being able to compile the Standard-recommended 6144 either) and I had a feeling there is a connection with the previous test case.

gcc	clang	msvc	intel
16384	16384	16384	4096

Initializer-clauses in one braced-init-list – initializerClauseInBracedInitList

Another test case which just repeatedly required generating an array with the required number of elements, and then iterating over it to sum up a value to get the required number for the test case.

Although the Standard recommends 16384 clauses, I found that no compiler had problems generating code for initializer lists of lengths up to 262144, which is several times the Standard-recommended value.

gcc	clang	msvc	intel
262144	262144	262144	262144

Scope qualifications of one identifier – scopeQualificationOfOneIdentifier

Although this seemed to be one of the more banal test cases, the higher values turned out to be fatal in the end to `clang` and `msvc` when I increased the bracket depth (via `-fbracket-depth=4096` for `clang`) but `gcc` was happy even with 4096 (this being 16 times the Standard-recommended value).

`clang` gives up somewhere at a value between 1024 and 2048 in a seemingly infinite recursive call between:

```
EmitTopLevelDecl(clang::Decl*)
```

and:

```
EmitDeclContext(clang::DeclContext const*)
```

but I'd rather say that 1024 scopes for a variable is more than enough.

`msvc` does not even support the Standard-recommended 256; it gives up at 128 with the error message:

```
scopeQualificationOfOneIdentifier-128.cpp(130):
fatal error C1061: compiler limit: blocks nested
too deeply
```

but with a depth set to 127 there were no problems.

`icc` had no problems with compiling to depths of 2048, but failed with 4096.

gcc	clang	msvc	intel
4096	1024	127	2048

Nested linkage-specifications – nestedLinkageSpecifiers

Personally, I think that nesting linkage specifications can, in the long term, lead to highly unmaintainable code. But if the Standard allows it, and there is even a recommended depth, who am I to protest. So, some code in the form of the one in Listing 11 (example shown for 4) was generated, and I have just acknowledged that 1024 sounds like a good number for this purpose unless you really want to be the source of future headaches.

`gcc` and `clang` had no issues compiling code with the aforementioned depth; however, the `msvc` compiler gave up somewhere at a value between 736 and 752 with the following error but works for 736:

```
nestedLinkageSpecifiers-752.cpp(748): fatal error
C1026: parser stack overflow, program too complex
```

`icc` accepted for this test case the Standard-recommended 1024.

gcc	clang	msvc	intel
1024	1024	736	1024

```
#include <iostream>

extern "C" { int fC() { return 1; }
extern "C++" { int fCx() { return 1; }
  extern "C" { int fCxC() { return 1; }
  extern "C++" { int fCxCx() { return 1; }
    int fun() {
      return 0+fC()+fCx()+fCxC()+fCxCx();}
    }
  }
 }
}
int main() {
  std::cout << fun() << std::endl;
}
```

Listing 11

```
#include <iostream>
constexpr unsigned long long sum(unsigned long
long n, unsigned long long s=0) {
  return n ? sum(n-1,s+n) : s;
}
constexpr unsigned long long k = sum(512);

int main() {
  std::cout << k<<std::endl;
}
```

Listing 12

```
#include <iostream>

template<int N0,int N1,int N2,int N3>
struct C {
  static const int v = N0 + N1 + N2 + N3;
};
int main() {
  C<1,1,1,1> c;
  std::cout << c.v << std::endl;
}
```

Listing 13

Recursive constexpr function invocations – recursiveConstexpr

Recursive **constexpr** function are not the most frequent ones; however, they can come in very handy from time to time. This test case required the application in Listing 12, where 512 is the actual required depth of the recursion. Running this test case gave the results that follow.

clang gave a very correct assessment of the situation:

```
recursiveConstexpr-512.cpp:5:30: error: constexpr
variable 'k' must be initialized by a constant
expression
constexpr unsigned long long k = sum(512);
                             ^   ~~~~~~~~
recursiveConstexpr-512.cpp:3:13: note: constexpr
evaluation exceeded maximum depth of 512 calls
```

gcc also recognized the situation, in the form of a warning message like:

```
recursiveConstexpr-512.cpp:5:41: error:
'constexpr' evaluation depth exceeds maximum of
512 (use '-fconstexpr-depth=' to increase the
maximum)
5 | constexpr unsigned long long k = sum(512);
```

However, after applying the suggested **-fconstexpr-depth=513**, it actually managed to compile the code, and by making that change we can bring the level of recursion up to 16384. At 32768, **gcc** also decided it was time to give up:

```
g++: internal compiler error: Segmentation fault
signal terminated program cc1plus
```

icc did not like 512 but worked nicely with 256.

msvc correctly recognized the scenario:

```
recursiveConstexpr-512.cpp(5): error C2131:
expression did not evaluate to a constant

recursiveConstexpr-512.cpp(3): note: failure was
caused by evaluation exceeding call depth limit
of 512 (/constexpr:depth<NUMBER>)
```

After specifying the required depth, the **msvc** compiler choked at 16384, 8192, 4096 in the form of an **Internal Compiler Error** but successfully compiled for 2048.

gcc	clang	msvc	intel
16384	16384	2048	256

Full-expressions evaluated within a core constant expression – fullExpressionInAConst

The value recommended for this situation is simply so huge (1048576) that I did not consider it necessary to increase it. The application generated for this case is just a simple addition of ones, being assigned to a constant value. Compiling a test case takes a long time, but the only compiler tested that had any problems with it was **msvc**, which gave up at somewhere a value between 65536 and 131072.

gcc	clang	msvc	intel
1048576	1048576	65536	1048576

Template parameters in a template declaration – templateParametersInTemplateDeclaration

This test case consisted of creating a source file along the lines of the one in Listing 13.

No compiler, except **msvc**, had problems compiling code with values up to 16384, which is 4 times the value recommended by the Standard. Just a small interesting observation is that while **gcc** generally outperformed all the other compilers (that were run on the same platform) from the point of view of speed, this test case was aced by **clang**, which delivered blazing fast speed for this test case, easily outperforming all the other compilers.

msvc failed 16384 with **fatal error C1111: too many template parameters** but in the end it managed to compile the Standard-recommended 1024.

gcc	clang	msvc	intel
16384	16384	1024	16384

Recursively nested template instantiations – recursivelyNestedTemplateInstantiations

The application in Listing 14 actually gave a headache to a few compilers. It seems that **icc** can handle recursively nested templates in a very predictable way:

```
recursivelyNestedTemplateInstantiations-
1024.cpp(9): error: excessive recursion at
instantiation of class "C<524>"`.
```

The troubles for **icc** were not over since – after a period of experimentation – I discovered that the maximum value it supports is 500. For values above 500, I get the previous strange error, with the value being always the test value - 500. So for 501 the error is: **error: excessive recursion at instantiation of class "C<1>"**. Strange, but interesting.

```
#include <iostream>
template<typename T>
struct B {
  typedef T BT;
};
template<int N>
struct C {
  typedef typename B<typename C<N-1>::T>::BT T;
};
template<>
struct C<0> {
  typedef int T;
};

int main()
{
  C<1024>::T c = 1024;
  std::cout << c << std::endl;
}
```

Listing 14

gcc also had its troubles:

```
recursivelyNestedTemplateInstantiations-
1024.cpp:9:45: fatal error: template
instantiation depth exceeds maximum of 900 (use
'-ftemplate-depth=' to increase the maximum)
```

but after specifying **-ftemplate-depth=1025** as an extra parameter, **gcc** succeeded. Interestingly, **gcc** expects +1 to the actual number.

clang aced this test, and without complaining compiled the entire 1024 iterations of template madness. An interesting side-note for **clang**: for 16384 it gave me the hint to use **-ftemplate-depth=16384** and then it gave me the warning in Listing 15. I had never seen this before, and my admiration for compiler writers has just gone up. But **gcc** compiled 16384 too, without this warning (I just had to specify **-ftemplate-depth=16385** as an extra parameter).

1024 proved to be fatal for **msvc**:

```
recursivelyNestedTemplateInstantiations-
1024.cpp(9): fatal error C1202: recursive type or
function dependency context too complex
```

Finally, it managed to compile 128.

gcc	clang	msvc	intel
16384	16384	128	500

Handlers per try block – handlersPerTryBlock

This test case consisted of generating a number of classes derived from **std::exception** that will act as objects to be thrown, then throwing an object of that kind in a **try** block and then writing a long list of **catch** statements for each class. No compiler had problems with code that contained 256 different handlers for a **try** block, as per the Standard-recommended value.

gcc	clang	msvc	intel
256	256	256	256

Number of placeholders – numberOfPlaceholders

This is not specifically a compiler limit and is more a library feature, but in the end we have to agree that all the compilers tested had an upper limit of 29, except **msvc** which draws the upper limit at 20.

gcc	clang	msvc	intel
29	29	20	29

Conclusion

Before you jump ship and decide that, based on these results, it's time to ditch your current compiler and switch to a different one, a big warning for you: don't. These test cases were specifically engineered for a unique purpose, and they are not real life situations. If they were, then maybe it would be time to rethink your source strategy.

Each of these compilers is able to perform adequately for any project you can find on the market today, and the purpose of this test was not to find a winner, but to see which does what well, and what improvements should be made for future releases.

Each of the tested compilers shines in some areas and performs poorly in others, and what follows are just a few (personal) observations. If you run the test cases, you will possibly reach a different conclusion.

gcc and **msvc** are the oldest of the tested bunch. Their age has positively affected their performance. Both of them are blazingly fast in all areas of compilation. **msvc** has a set of limitations, that you will not notice in your average daily programming routine unless you specifically look for them, while **gcc** can compile basically everything that you throw at it, assuming you have the patience to wait for the compilation time of a large code base, and your computer can cope with the expectations of the compiler.

icc, which came more than 20 years after **msvc**, promises faster-than-average code targeting its own processors, good c++17 support and also a decent speed. Sadly, it is packaged into a suite downloadable on a trial basis from Intel's homepage, and this possibly makes hobbyist programmers or the advocates of open source stay away unless forced by some specific requirements.

clang is the newcomer and the youngest of the tested compilers. It outperforms all the others when it comes to more recent C++ features but seems to struggle with notions and constructs that other compilers have had a few extra decades to polish till perfection. But the speed at which the community picked it up, and made it into one of the most used compilers today hints at a bright future for this product. ■

References

[ANNEX-B] 'Annex B: (normative) Implementation quantities' of the C++ Standard: https://eel.is/c++draft/implimits

[CLANG] Clang limites: https://clang.llvm.org/docs/UsersManual.html#controlling-implementation-limits

[GCC] gcc limits: https://gcc.gnu.org/onlinedocs/gcc-9.2.0/cpp/Implementation-limits.html

[GITHUB] The test code: https://github.com/fritzone/cpp-stresstest

[MSVC] Microsoft compiler limits: https://docs.microsoft.com/en-us/cpp/cpp/compiler-limits?view=vs-2019

```
warning: stack nearly exhausted; compilation time may suffer, and crashes due to stack overflow are likely
[-Wstack-exhausted]
    typedef typename B<typename C<N-1>::T>::BT T;
                                ^
recursivelyNestedTemplateInstantiations-1024.cpp:9:30: note: in instantiation of template class 'C<15286>'
```

Listing 15

Afterwood

Assume failure by default. Chris Oldwood considers various fail cases.

Two roads diverged in a wood, and I –
I took the one less travelled by,
And that has made all the difference.
~ Robert Frost

Despite being written in the early part of the 20th century, I often wonder if Robert Frost's famous poem might actually have been about programming. Unless you're writing a trivial piece of code, every function has a happy path and a number of potential error paths. If you're the optimistic kind of programmer, you'll likely take the well trodden path and focus on the happy outcome and hope that no awkward scenarios turn up. This path is exemplified in the original version of that classic first program which displays "hello, world!" on the console:

```
main()
{
  printf("hello, world\n");
}
```

A standards-conforming version of this classic C program requires the **`main`** function to be declared with an **`int`** return type to remind us that we need to inform the invoker of any problems, but luckily we get to remain optimistic as we can elide any return value (only for a function named **`main`**) and happily accept the default – **`0`**. Consequently, irrespective of whether or not the **`printf`** statement actually works, we're going to tell the caller that everything was hunky-dory.

The classic C version relies on a spot of 'legal slight-of-hand' to allow you to put the program's return value out of mind whereas C# and Java needed to find another way to let you ignore it so they allow you to declare **`main`** without a return type at all:

```
public class Program
{
  public static void Main(...)
  {
    // print "Hello World!"
  }
}
```

Of course, these languages use exceptions internally to signal errors so it doesn't matter, right? Well, earlier versions of Java would return **`0`** if an uncaught exception propagated through **`main`** so you can't always rely on the language runtime to act in your best interests [Wilson10]. Even with .Net you can experience some very negative exit codes when things go south which will make a mockery of that tried-and-tested approach to batch-file error handling everyone grew up with:

```
IF ERRORLEVEL 1
```

Unless you know that **`Main`** can *also* return an exit code I don't think it should be that surprising that people have resorted to silencing those pesky errors with an all-encompassing **`try`**/**`catch`** block:

```
public static void Main(...)
{
  try
  {
    // Lots of cool application logic.
  }
  catch
  {
    // Write message to stderr.
  }
}
```

I wonder if this pattern is more common than even I've experienced as PowerShell has taken the unconventional approach of treating *any* output on the standard error stream as a sign that a process has failed in some way. This naturally causes a whole different class of errors on the other side of the fence that could be considered worse than 'the cure'.

Back in the world of C and C++ we can be pro-active and acknowledge our opportunity to fail but are we still being overly optimistic by starting out by assuming success?

```
int main(int argv, char **argv)
{
  int result = EXIT_SUCCESS;
  // Lots of cool application logic.
  return result;
}
```

It's generally accepted that small focused functions are preferable to long rambling ones but it's still not that uncommon to need to write some non-trivial conditional logic. When that logic changes over time (in the absence of decent test coverage) what are the chances of a false positive? When it comes to handling error paths, I'd posit that it's categorically *not* zero.

The trouble with error paths are that they are frequently less travelled and therefore far less tested. A bug in handling errors where the flow of control is not correctly diverted can lead to other failures later on which are then harder to diagnose as you'll be working on the assumption of some earlier step having completed successfully. In contrast, a false negative should cause the software to fail faster which may be easier to diagnose and fix. To wit, assume failure by default:

```
int result = EXIT_FAILURE;
```

The term 'defensive programming' is one which was well intentioned, and requires an acknowledgement of failure to allow robust code to be written, but it has also

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the lounge below his bedroom. With no Godmanchester duck race to commentate on this year, he's been even more easily distracted by messages to gort@cix.co.uk or @chrisoldwood

Every test framework I've encountered makes the optimistic assumption that as long as your code doesn't blow up, then it's correct

been used to cover a multitude of sins – counter-intuitively making our lives harder, not easier. It stems from a time when development cycles were long, releases were infrequent, and patching was expensive. In a modern software delivery process Mean Time to Recovery is often valued more highly than Mean Time to Fail.

Another area where I find an overly optimistic viewpoint is with test frameworks. Take this simple test, which does nothing:

```
[Test]
public void   doing_nothing_is_better_than_being_
busy_doing_nothing()
{
}
```

Plato once said that an empty vessel makes the loudest sound, and yet a test which makes no assertions is usually silent on the matter. Every test framework I've encountered makes the optimistic assumption that as long as your code doesn't blow up, then it's correct, irrespective of whether or not you've even made any attempt to assert that the correct behaviour has occurred. This is awkward because forgetting to finish writing the test (it happens more often than you might think) is indistinguishable from a passing test.

When practising TDD, the first step is to write a failing test. This is not some form of training wheels to help you get used to the process, it's fundamental in helping you ensure that what you end up with is working code and test coverage for the future. Failing by default brings clarity around what it means to succeed, or in a modern agile parlance – what is the definition of 'done'?

In those very rare cases where the outcome cannot be expressed as the absence of some specific operation occurring, there are always the following constructs to make it clear to the reader that you didn't just forget to finish writing the test:

```
Assert.Pass();
Assert.That(. . ., Throws.Nothing);
```

The few mocking frameworks which I've had the displeasure to use also have a similar misguided level of optimism when it comes to writing tests – they try really hard to hide dependencies and just make your code work, i.e. they adopt the classic 'defensive programming' approach which I mentioned earlier. It's misguided because exposing your dependencies to the reader is a key part of illustrating what the reader needs to know to understand what interactions the code might rely on. If this task is onerous then that's probably a good sign you need to do some refactoring!

I'm being overly harsh on Hello World; it's a program intended for educational purposes not a shining example of 100% error-free code (whatever that means). I'm sure a kitten dies every time an author writes 'error handling elided for simplicity' but maybe that's an unavoidable cost of trying to present a new concept in the simplest possible terms. However, when it comes to matters of correctness perhaps we need to take the difficult path if we are going to provide the most benefit in the longer term. ■

Reference

Matthew Wilson (2010) 'Quality Matters #6: Exceptions for Practically-Unrecoverable Conditions' in *Overload* #99, October 2010, available at: https://accu.org/index.php/journals/1706

Join ACCU
visit
www.accu.org
for details

We need your help!

ACCU is a volunteer organisation. Without volunteers, we cannot function. We need:

- Volunteers for vacant posts on the committee
- People to write articles (regularly or occasionally)
- People who can help out with particular short-term and long-term projects

If you would like to help but are not sure how you can – you may not have a lot of time, or may not be able to commit to anything long-term – please get in touch. You may have just the skills we need for

"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

intel Software

PURE CODE ADRENALINE

Accelerate applications for enterprise, cloud, HPC, and AI.

Develop high-performance parallel code and accelerate workloads across enterprise, cloud, high-performance computing (HPC), and AI applications.

Amp up your code: **www.qbssoftware.com**

For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.

Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.
© Intel Corporation