Supporting document for Hive proposal 1: outreach for evidence of container-style use in industry

Audience: LEWG, SG14, WG21
Document number: P3011R0
Date: 2023-10-15
Project: Supporting document for std::hive proposal 1
Reply-to: Matthew Bentley mattreecebentley@gmail.com

Background

While personally I have had multiple contacts with people in various professions over the past 7 years who have been using this style of container, a lot of that stuff is in emails long-since deleted or in old abandoned accounts. Many members of SG14 and WG21 have stated explicitly in LEWG and SG14 meetings that they use this style of container, but for reasons unknown, some members remain unconvinced.
Anyway, this is the result of a brief and not extensive outreach to solicit feedback on use of this style of container in industry.

First outreach: ~200 emails sent out to various businesses, none of which I have contacted before. ie. Cold-calling. Full list at end of document.

Second: Posting on reddit and discord.

Third: Posting on TIGsource, an old (but less-active nowadays) independent gamedev website.

Questions asked

While the form differed depending on the company/individual I was contacting, the general form was as follows:
“To whoever it may concern,
I've been asked by the C++ standards committee to gather evidence for use of a specific type of technology in industry, such that we can justify standardising it.
To that end I am contacting businesses which may employ the same or similar techniques. I would greatly appreciate it if you can pass the following quick questions onto a lead engineer in your company. In the interests of proving my identity, please see the email address on the following C++ standards paper:

Questions for engineers:
1. Are you using the following type of data container in your work: sequential storage, either in multiple memory blocks or one singular memory block, elements are marked by a boolean/bit/token when erased and these marked elements are subsequently skipped over during iteration.
2. If so, is the container multiple-memory-block or singular? What do you use the container for (if allowed to say)?

3. Are you aware of plf::colony and the corresponding std::hive proposal for the standard library?
4. Do you think that there would be benefits to this type of container being standardised?
5. Do you have any additional comments or questions?"

On discord and reddit I truncated these questions, as people are well aware of std::hive/colony. On TIGsource I forgot to post 3.

Results, tallied

Over email 8 businesses/individuals responded with answers (Radiant Nuclear, Tommy Refenes of Team Meat, Interance, Mind.be, Essensium, Supergiant Games, StellarScience). Their exact responses are detailed at the end of this document. A few more responded but turned out not to be using C/C++ in their frameworks.

Over discord 4 people responded, and on reddit 11 people responded to my questions (many others responded, but did not answer my questions). On TIGsource 1 person responded. Screenshots are at the end of this document.

Answers from all sources above will be counted as follows:

"1. Are you using the following type of data container in your work: sequential storage, either in multiple memory blocks or one singular memory block, elements are marked by a boolean/bit/token when erased and these marked elements are subsequently skipped over during iteration."

email: 5/7 yes
discord: 2/4 yes
reddit: 6/11 yes
TIGsource: 1/1 yes

Total: 61% yes

It should be noted that many respondents were utilizing sub-optimal strategies here, such as unrolled linked lists, or having a vector of pointers to dynamically-allocated elements (same/worse performance as a linked list). Some people, misunderstanding the aim of this approach, stated they were swapping-and-popping with the back element (which invalidates pointers/iterators to each back element as it is moved).

"2. If so, is the container multiple-memory-block or singular?"

Obviously only those who responded yes could answer this one, and not everyone did (even if they said yes).

Email: 3 multiple, 2 singular, 1 both.
Discord: no responses
Reddit: 1 multiple, 3 singular, 1 both.
TIGsource: 1 singular.

"3. Are you aware of plf::colony and the corresponding std::hive proposal for the standard library?"

email: 4 yes, 3 no.

On discord and reddit I didn’t bother with this and the following question as most were likely to know about the hive proposal. However a discord user noted that they switched from a custom implementation to plf::colony for a project several years ago and got a 10-15% performance increase.

“4. Do you think that there would be benefits to this type of container being standardised?”

email: 6 yes, 1 yes but in the future.
TIGsource: 1 no opinion.

Summary

While my sample size is small, the data fits with my previous experience and observations that this style of container is common in industry use.

Responses

Note: a few lines which might be considered sensitive/non-disclosable information have been redacted from some of the responses.

Email:

Team Meat:

"Hey, thanks for reaching out. I hope my answers help!

1. Kinda but not often
2. What we do use is multiple memory block for the most part.
3. It's all sort of custom for us, there isn't really a one size fits all solution
4. I read the proposal and it seems like it's pretty cool
5. I could see the benefits to it being standardized, but I probably wouldn't use it. I have a lot of stuff in engine that already works and I don't think I would port it and if I needed something new I would just use what I already have.

I'm not the best person to ask about this...I wrote my own version of std::list decades ago because I thought it was fun to do and it taught me about templates. That implementation is in everything I make now haha." - Tommy Refenes

RadiantNuclear:

"Hi Matt,

Thanks for reaching out, happy to offer some advice, our use case for C++ is split between two domains, simulation software for desktop and embedded software for bare metal applications.

Yes

Singular

N/A

No

Yes

I would like to see it also paired with a fixed width type similar to boost::container::static_vector or the fixed_capacity_vector proposal. These sorts of fixed type representations are extremely important for embedded / bare-metal / OS development applications and most projects in the space reimplement them. Closest thing in the embedded space to a usable standard library is ETL: https://www.etlcpp.com/ I’ve ended up rebuilding a lot of the same container types multiple times for different embedded projects, and our use case seems generally underrepresented or considered in the standards committee. I think the best evidence for this is the recent deprecation and subsequent undeprecation of compound assignments to volatile storage, which fundamentally broke multiple massive codebases I have designed or contributed to due to the nature of how memory mapped device drivers are implemented.

That’s all, happy to answer more questions in the future.

Bob Urberger

CTO"

Interance:

"Hi Matt,

1. No.

2. -

3. We do use containers such as std::list, but not because of the stable iterators. For example, our JSON implementation maps JSON arrays to lists with a monotonic buffer resource allocator to avoid re-allocations (that would punch holes into the consecutive buffer space).

4. I’m occasionally checking for updates on the standardization process and have seen hive/colony pop up before. But I haven’t looked at the paper in detail.

5. Definitely. While we don’t have an immediate use case right now, I think having a container like this would be great.

6. Keep up the good work! It takes a lot of dedication to get something standardized. The standard library needs people like you. :)

Best,

Dominik"

Mind.be:

"Hi Matt,

Thanks for consulting us, it's much appreciated! :-)

Here after, the quick answers:

1 - Rarely

2 - Both (multi and single memory blocks) / Cascading asynchronous tasks (detached threads, timeout-based cleaning, garbage-collector like...), Delayed operation on maybe-obsolete data, delayed or split image processing...

3 - Sometime array/vector + shared_ptr

4 - Now I am, thanks

5 - Probably not to incorporate into the standard soon, but definitely relevant in an extension library, especially relevant in boost. Since big data, IA, parallel calculation and async architectures are getting widespread, and more and more developers will face such contexts, this design is worth being available. But it probably better suits an external and/or dedicated library for now, since the API and complexity goes one step too far to be part of the coming standards.

6 - If standardized, please avoid diverging from the existing containers APIs, do not make the language yet more complex; it's already very difficult to keep beginners onboard!!! Take more time to propose something clean and stable, do not hurry up. Avoid API changes after release.

Kind Regards,

--

Francois Gerin "

Note: Francois was unaware of the long development time that has occurred.

Essensium:

"Hello Matt,

I've briefly discussed your questions with François and mostly share his points of view. (i.e. std::hive might be a too specific use case, there is a need to not complexify the stl if it can be avoided).

However since it's in preparation since a while now and backed by Nicolai Josuttis, I would say go for it.

Just my 2 cents.

BR,

Patrick Havelange "

Supergiant games:

"

Hi Matt,

Thanks for reaching out with your questions! My name is Nikola and I'm an engineer at Supergiant Games. I'd be glad to answers any questions you may have.

1. Are you using the following type of data container in your systems: sequential element storage, either in multiple memory blocks or one singular memory block, elements are marked by a boolean/bit/token when erased and these marked elements are subsequently skipped over during iteration (enables erasure without invalidating pointers to elements).

- In one place in our codebase, we use a very large std::array containing data structures that contain { actualData, NextFreeIndex }. We do not want to change the size of this container at runtime because we want to control memory use closely due to memory constraints on consoles. In all other places, we don't need stable iterators and we use a lot less memory, so we generally use classic arrays and vectors, but also hashmaps in multiple places.

2. If so, is the container multiple-memory-block or singular?

- Single memory block, since it's a std::array

4. Are you aware of the std::hive proposal for the standard library, or it's precursor plf::colony?

- I've investigated plf::colony a few years ago, it looked like an interesting proposal, however we didn't find much use in switching because our code base is already established

5. Do you think that there are benefits to this style of container being standardized?

- I definitely see its uses. There are two main barriers to its use (that probably apply to any new proposal): For established smaller game engines, like ours, it may not be worth it to switch to using std::hive, since we have smaller engineering teams and established low-level codebases; and most AAA studios have their own specialized containers and aren't relying on std. However, for new projects, I see the appeal of using a container like this.

6. Any additional comments or questions?

- My concern with using hive would be that it can balloon in memory use with a bunch of memory blocks with one a few active elements. Any use of this container would have to be carefully vetted and tested, so I wouldn't consider it a drop in replacement to any other container.

I'm open to any futher discussion on this topic, feel free to contact me

Best,

Nikola

"

Note: I have subsequently clarified to Nikola that the memory waste will only happen if there were many random erasures and no insertions (ie. a situation where blocks lose all but a few elements and therefore don't get freed) – in which case one would be in the same situation as when using vector and popping or erasing, till one calls shrink_to_fit.

StellarScience:

"Hi Matt,

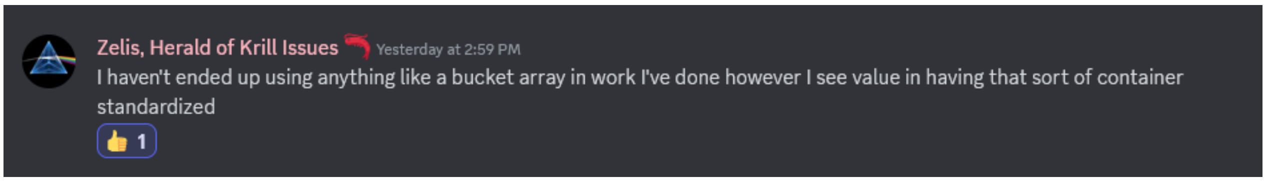
Sorry for the delay. Here's what I've gathered:

1. We do not have that type of container (or no one recognized anything as such).
2. N/A
3. There may be a few extremely isolated cases where we need stable iterators or pointers, but no one could point me to any current examples.
4. Yes, I am aware of hive.
5. Although we don't seem to use that kind of container, I could see others needing it.
6. No, but thank you!

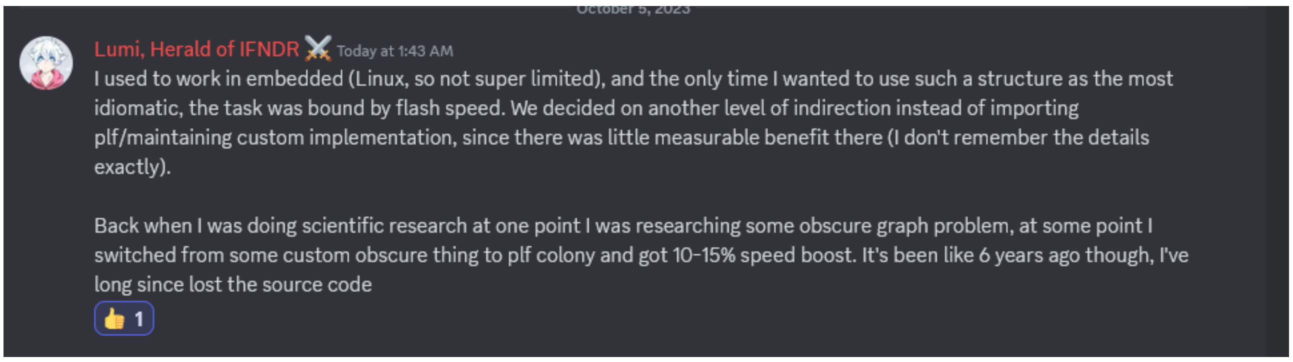
   K.R Walker

"

Discord:



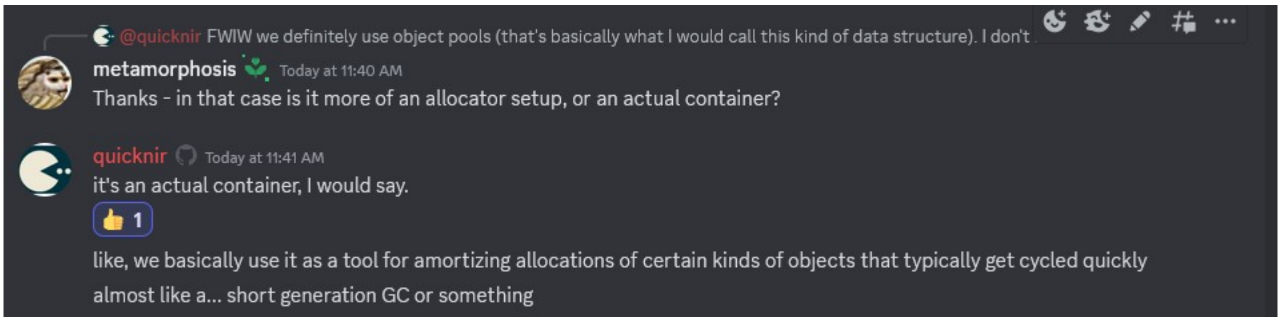
Zelis, Herald of Krill Issues Yesterday at 2:59 PM
I haven't ended up using anything like a bucket array in work I've done however I see value in having that sort of container standardized
👍 1



October 5, 2023

Lumi, Herald of IFNDR Today at 1:43 AM
I used to work in embedded (Linux, so not super limited), and the only time I wanted to use such a structure as the most idiomatic, the task was bound by flash speed. We decided on another level of indirection instead of importing plf/maintaining custom implementation, since there was little measurable benefit there (I don't remember the details exactly).

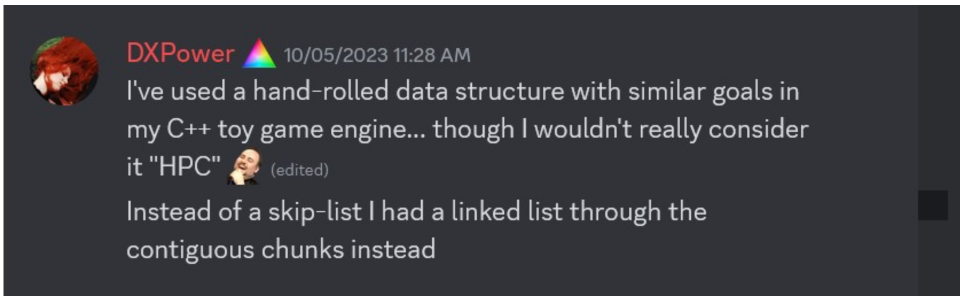
Back when I was doing scientific research at one point I was researching some obscure graph problem, at some point I switched from some custom obscure thing to plf colony and got 10-15% speed boost. It's been like 6 years ago though, I've long since lost the source code
👍 1



@quicknir FWIW we definitely use object pools (that's basically what I would call this kind of data structure). I don't

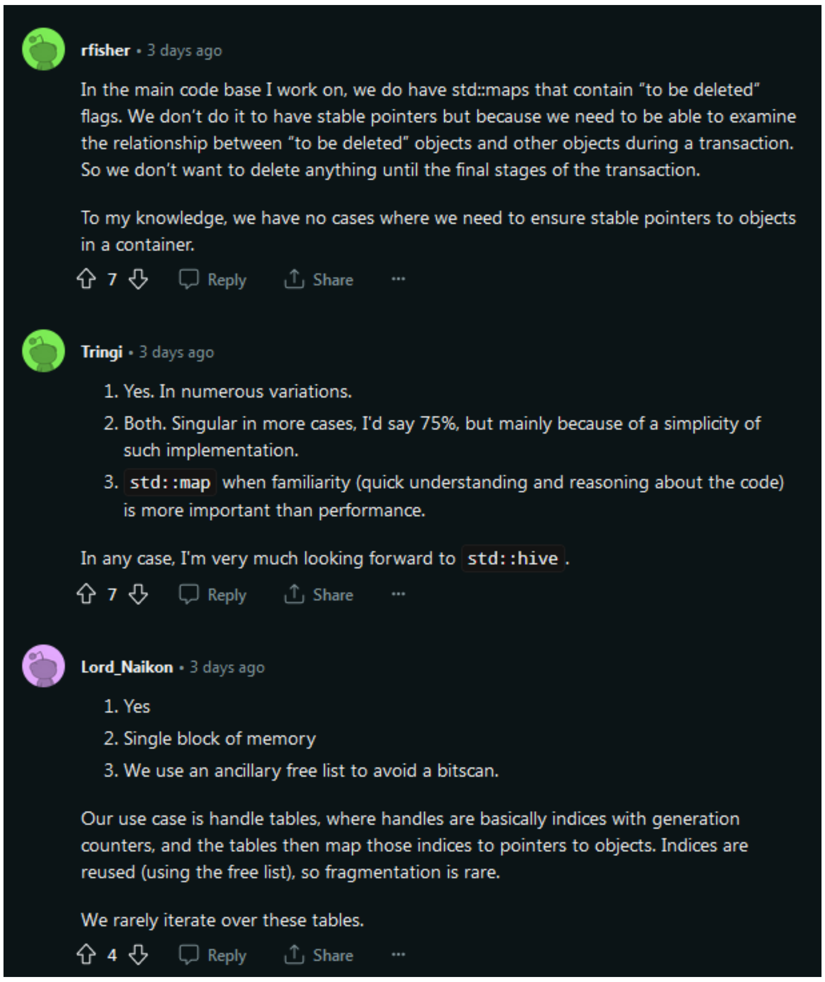
metamorphosis Today at 11:40 AM
Thanks - in that case is it more of an allocator setup, or an actual container?

quicknir Today at 11:41 AM
it's an actual container, I would say.
👍 1
like, we basically use it as a tool for amortizing allocations of certain kinds of objects that typically get cycled quickly
almost like a... short generation GC or something



DXPower 10/05/2023 11:28 AM
I've used a hand-rolled data structure with similar goals in my C++ toy game engine... though I wouldn't really consider it "HPC" (edited)
Instead of a skip-list I had a linked list through the contiguous chunks instead

Reddit:



rfisher • 3 days ago

In the main code base I work on, we do have std::maps that contain "to be deleted" flags. We don't do it to have stable pointers but because we need to be able to examine the relationship between "to be deleted" objects and other objects during a transaction. So we don't want to delete anything until the final stages of the transaction.

To my knowledge, we have no cases where we need to ensure stable pointers to objects in a container.

Tringi • 3 days ago

1. Yes. In numerous variations.
2. Both. Singular in more cases, I'd say 75%, but mainly because of a simplicity of such implementation.
3. `std::map` when familiarity (quick understanding and reasoning about the code) is more important than performance.

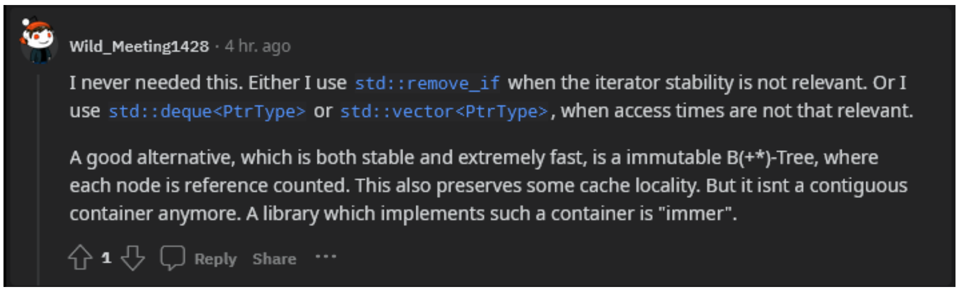
In any case, I'm very much looking forward to `std::hive`.

Lord_Naikon • 3 days ago

1. Yes
2. Single block of memory
3. We use an ancillary free list to avoid a bitscan.

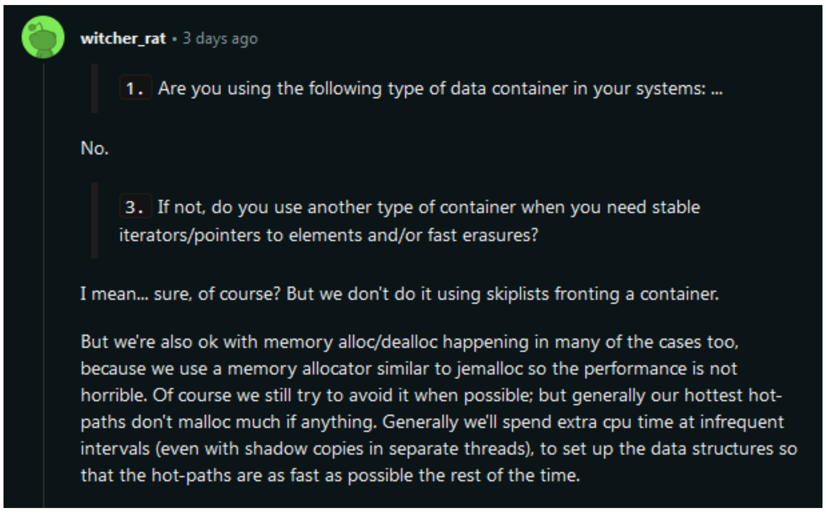
Our use case is handle tables, where handles are basically indices with generation counters, and the tables then map those indices to pointers to objects. Indices are reused (using the free list), so fragmentation is rare.

We rarely iterate over these tables.


Wild_Meeting1428 • 4 hr. ago

I never needed this. Either I use `std::remove_if` when the iterator stability is not relevant. Or I use `std::deque<PtrType>` or `std::vector<PtrType>`, when access times are not that relevant.

A good alternative, which is both stable and extremely fast, is a immutable B(+*)-Tree, where each node is reference counted. This also preserves some cache locality. But it isnt a contiguous container anymore. A library which implements such a container is "immer".


witcher_rat • 3 days ago

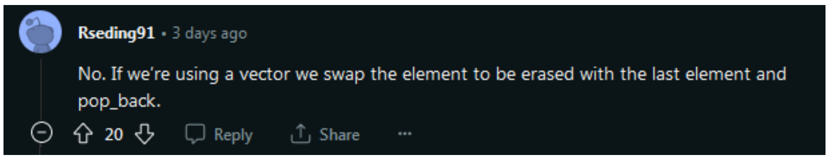
> 1. Are you using the following type of data container in your systems: ...

No.

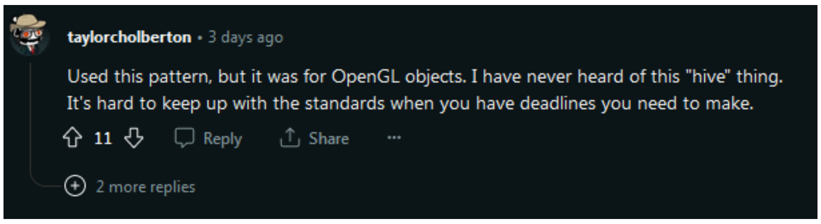
> 3. If not, do you use another type of container when you need stable iterators/pointers to elements and/or fast erasures?

I mean... sure, of course? But we don't do it using skiplists fronting a container.

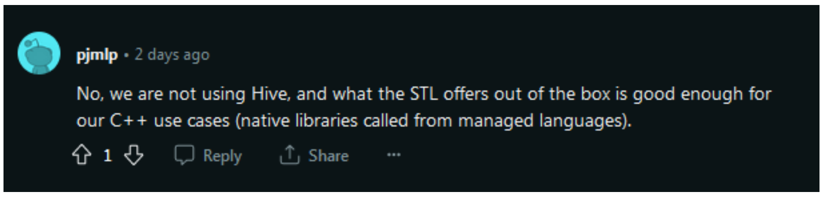
But we're also ok with memory alloc/dealloc happening in many of the cases too, because we use a memory allocator similar to jemalloc so the performance is not horrible. Of course we still try to avoid it when possible; but generally our hottest hot-paths don't malloc much if anything. Generally we'll spend extra cpu time at infrequent intervals (even with shadow copies in separate threads), to set up the data structures so that the hot-paths are as fast as possible the rest of the time.


Rseding91 • 3 days ago

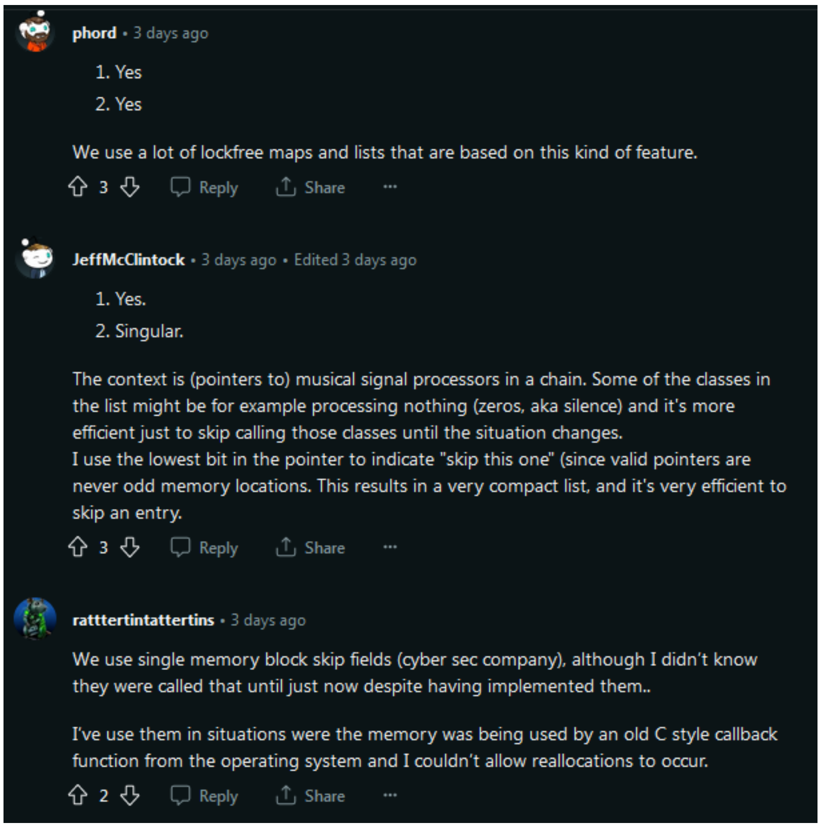
No. If we're using a vector we swap the element to be erased with the last element and pop_back.


taylorcholberton • 3 days ago

Used this pattern, but it was for OpenGL objects. I have never heard of this "hive" thing. It's hard to keep up with the standards when you have deadlines you need to make.


pjmlp • 2 days ago

No, we are not using Hive, and what the STL offers out of the box is good enough for our C++ use cases (native libraries called from managed languages).



phord • 3 days ago

1. Yes
2. Yes

We use a lot of lockfree maps and lists that are based on this kind of feature.

3 Reply Share ···

JeffMcClintock • 3 days ago • Edited 3 days ago

1. Yes.
2. Singular.

The context is (pointers to) musical signal processors in a chain. Some of the classes in the list might be for example processing nothing (zeros, aka silence) and it's more efficient just to skip calling those classes until the situation changes.
I use the lowest bit in the pointer to indicate "skip this one" (since valid pointers are never odd memory locations. This results in a very compact list, and it's very efficient to skip an entry.

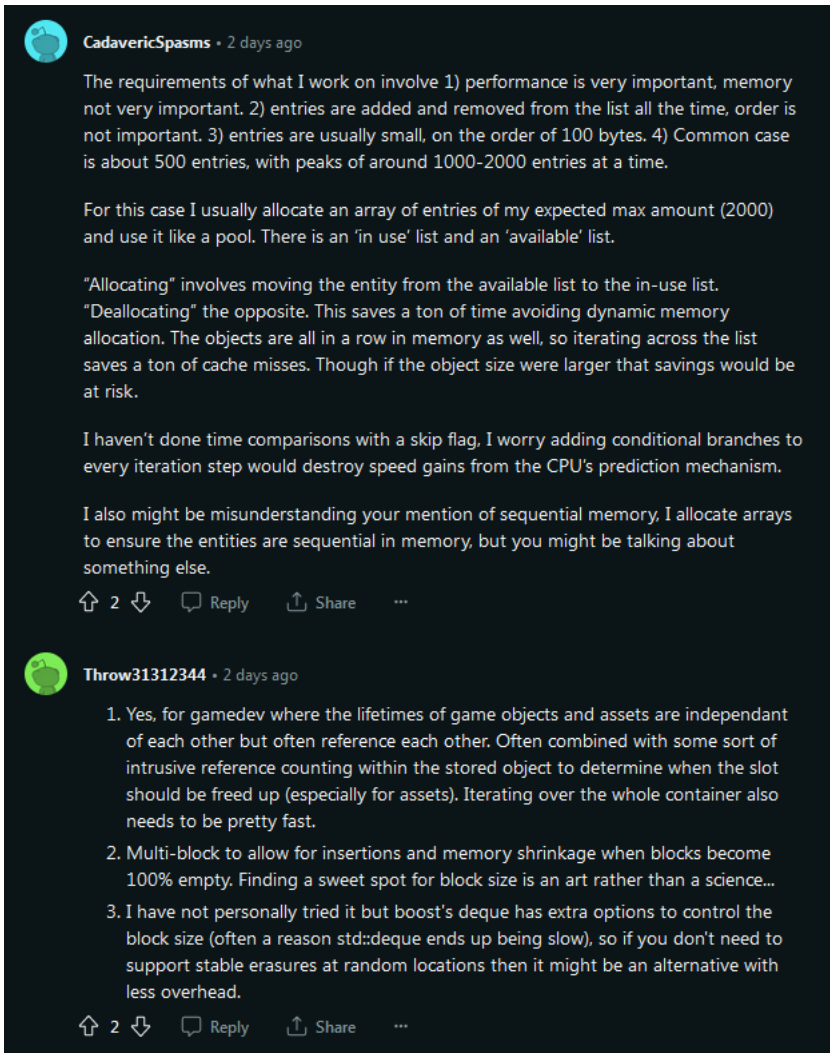
3 Reply Share ···

ratttertintattertins • 3 days ago

We use single memory block skip fields (cyber sec company), although I didn't know they were called that until just now despite having implemented them..

I've use them in situations were the memory was being used by an old C style callback function from the operating system and I couldn't allow reallocations to occur.

2 Reply Share ···



CadavericSpasms • 2 days ago

The requirements of what I work on involve 1) performance is very important, memory not very important. 2) entries are added and removed from the list all the time, order is not important. 3) entries are usually small, on the order of 100 bytes. 4) Common case is about 500 entries, with peaks of around 1000-2000 entries at a time.

For this case I usually allocate an array of entries of my expected max amount (2000) and use it like a pool. There is an 'in use' list and an 'available' list.

"Allocating" involves moving the entity from the available list to the in-use list. "Deallocating" the opposite. This saves a ton of time avoiding dynamic memory allocation. The objects are all in a row in memory as well, so iterating across the list saves a ton of cache misses. Though if the object size were larger that savings would be at risk.

I haven't done time comparisons with a skip flag, I worry adding conditional branches to every iteration step would destroy speed gains from the CPU's prediction mechanism.

I also might be misunderstanding your mention of sequential memory, I allocate arrays to ensure the entities are sequential in memory, but you might be talking about something else.

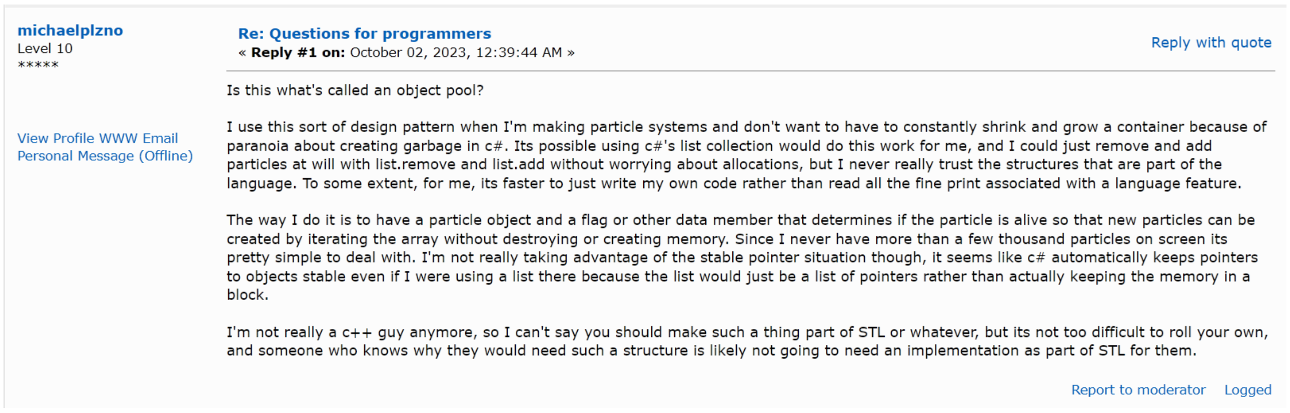
2 Reply Share ···

Throw31312344 • 2 days ago

1. Yes, for gamedev where the lifetimes of game objects and assets are independant of each other but often reference each other. Often combined with some sort of intrusive reference counting within the stored object to determine when the slot should be freed up (especially for assets). Iterating over the whole container also needs to be pretty fast.
2. Multi-block to allow for insertions and memory shrinkage when blocks become 100% empty. Finding a sweet spot for block size is an art rather than a science...
3. I have not personally tried it but boost's deque has extra options to control the block size (often a reason std::deque ends up being slow), so if you don't need to support stable erasures at random locations then it might be an alternative with less overhead.

2 Reply Share ···

TIGsource:


michaelplzno
Level 10
*****

View Profile WWW Email
Personal Message (Offline)

Re: Questions for programmers
« **Reply #1 on:** October 02, 2023, 12:39:44 AM »

Reply with quote

Is this what's called an object pool?

I use this sort of design pattern when I'm making particle systems and don't want to have to constantly shrink and grow a container because of paranoia about creating garbage in c#. Its possible using c#'s list collection would do this work for me, and I could just remove and add particles at will with list.remove and list.add without worrying about allocations, but I never really trust the structures that are part of the language. To some extent, for me, its faster to just write my own code rather than read all the fine print associated with a language feature.

The way I do it is to have a particle object and a flag or other data member that determines if the particle is alive so that new particles can be created by iterating the array without destroying or creating memory. Since I never have more than a few thousand particles on screen its pretty simple to deal with. I'm not really taking advantage of the stable pointer situation though, it seems like c# automatically keeps pointers to objects stable even if I were using a list there because the list would just be a list of pointers rather than actually keeping the memory in a block.

I'm not really a c++ guy anymore, so I can't say you should make such a thing part of STL or whatever, but its not too difficult to roll your own, and someone who knows why they would need such a structure is likely not going to need an implementation as part of STL for them.

Report to moderator Logged

Text reproduced as image is small to fit on page:

“Is this what's called an object pool?

I use this sort of design pattern when I'm making particle systems and don't want to have to constantly shrink and grow a container because of paranoia about creating garbage in c#. Its possible using c#'s list collection would do this work for me, and I could just remove and add particles at will with list.remove and list.add without worrying about allocations, but I never really trust the structures that are part of the language. To some extent, for me, its faster to just write my own code rather than read all the fine print associated with a language feature.

The way I do it is to have a particle object and a flag or other data member that determines if the particle is alive so that new particles can be created by iterating the array without destroying or creating memory. Since I never have more than a few thousand particles on screen its pretty simple to deal with. I'm not really taking advantage of the stable pointer situation though, it seems like c# automatically keeps pointers to objects stable even if I were using a list there because the list would just be a list of pointers rather than actually keeping the memory in a block.

I'm not really a c++ guy anymore, so I can't say you should make such a thing part of STL or whatever, but its not too difficult to roll your own, and someone who knows why they would need such a structure is likely not going to need an implementation as part of STL for them.“

Note: I have subsequently explained the disadvantages of this person’s approach to them (branching, non-O(1) iteration, wasted memory etc)

List of businesses contacted over email

reddit cpp jobs listings:

Nokia
https://www.missionbuddies.de/

Freeform.co

qt.io

nyriad

Tenzir

stellarscience

Otherside Entertainment

Objectbox

Soundradix

Interance

rev.ng

Johnson and Johnson

severalnines

guardsquare

wolverine trading

sonarsource

chess.com

zivid

ARA

IAR

mind.be

Atomos space

California Technical Media

img.ly

google, robotics:

Shield AI
Graymatter robotics

Cruise Cars

Magna International

May Mobility

Apptronik

Relativity space

Spaceex

Sierra Space

Honeybee robotics

SCythe Robotics

Arrow Electronics

google, HFT:

virtu financial
2sigma
tower research

Jump trading

DRW

Hudson river trading

xtx markets

Flow traders

Optiva

IMC

Xr TRADERS

beacon

seldon.io

Robinhood

Lykke

Turbo power systems

Lonk

Celoxica

Springdew

OA Systems

PrimeXM

Xcell

google, gaming:

aaa studios
kevuru
double fine

toys for bob

Mike Acton (Unity)

Harebrained schemes

Obsidion entertainment

Thunder lotus

Epic Games

Poncle

Tencent

CD project red

Grinding gear games

Croteam

Mundfish

Bioware

4A games

Yacht Club Games

Playdead

google, HPC/AI:

IBM
DDN
Rescale

AdvancedHPC

Penguin Computing

Nesi (Niwa)

Monad

Atom Computing

Qindra

Qblocks.cloud

Fluidstack

Anabrid

Universal Quantum

Ampd

Kaleidosim

HRL

Riverlane

Tachyum

sima.ai

meetiqm

Optalysis

Altair

Kitware

ciq.co

GRCooling

Hadean

tech-x

SiPearl

Blueshift memory

Kalray

Ceremorphic

Cascade Technologies

Avicena

google, physics:

https://www.sibyllabiotech.it/
Ansys
Arete

Radiant Nuclear

TWI UK

Kuano

Brelyon

Metron Science

First Light Fusion

Sarcos

CFD research

BAE Systems

google, random:

JpMorgan and Chase
JK Barnes
Amentum

Chevron

Akuna Capital

Intelliswift

Rocketlab NZ

GSC-games

Team Meat

Terry Cavanagh

Thunderful games

Black Salt Games

Frictional games

Supergiant games

Sorath Games

Gearbox Games

Petroglyph games

Nicalis

Joymasher

Daniel Mullins Games

Aminita design

Flying Wild Hog

Zojio

Aceteam

Yager games

Increpare

Dinosaur Polo Club

Derek Yu

Heart Machine

That game company

Innersloth

Extremely OK games

Eric Barone

Studio MDHR

Hello Games

Night School Studio

Giant Squid Games