The Performance Price of Dynamic Memory in C++

Ivica Bogosavljevic

C++ now

2021
MAY 2-7
Aspen, Colorado, USA

About me

- Ivica Bogosavljevic - application performance specialist
- Professional focus is application performance improvement - techniques used to make your C/C++ program run faster by using better algorithms, better exploiting the underlying hardware, and better usage of the standard library, programming language, and the operating system.
- Writer for software performance blog: Johny's Software Lab - link in the footer
- Work as a performance consultant - helping developer teams deliver fast software

Introduction

- Two types of programs when it comes to memory usage
  - All allocations are a few large blocks of memory
    - Data structures such as arrays, vectors and matrices typically need one allocation to keep all their internal data
    - Examples: Image, audio and video processing algorithms typically require a few large buffers to hold the data
  - The program allocates many blocks during the program lifetime
    - Fast random access data structures (e.g. trees or hash maps) allocate many small memory chunks
    - C++ vectors that stores pointers (std::unique_ptr<>, std::shared_ptr<> and their relatives in a container
    - Classes that have pointer, std::unique_ptr<> and std::shared_ptr<> members
- Programs that allocate memory, deallocate memory or access memory in a random access fashion can suffer from performance degradation

Introduction

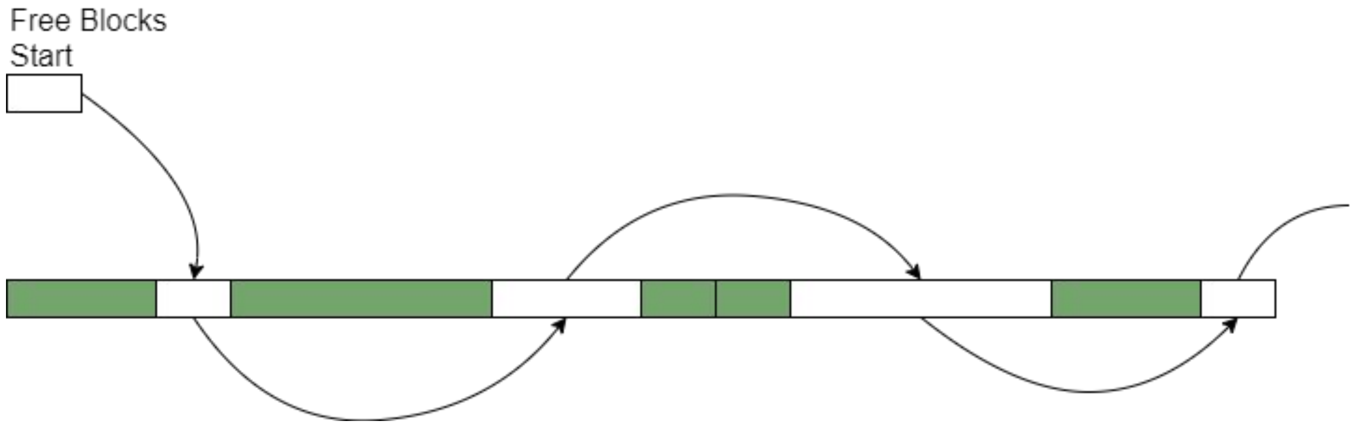
- Why is your program that uses dynamic memory slow?
- If you allocate and deallocate memory in many small chunks, performance of *malloc (new)* and *free (delete)* can be the bottleneck
  - This will typically be visible in profiler output: your program will spend a lot of time in *malloc* and *free* functions
- If you access your data structure in a random access fashion (tree, hash map, allocated objects, etc.), performance will suffer due to data cache misses
  - This will not show easily in the simple profiler output, but there are profilers that can measure data cache misses, e.g. *perf stat* or *cachegrind*
  - More information on data caches a bit later

Johny’s Software Lab

Performance of memory allocation

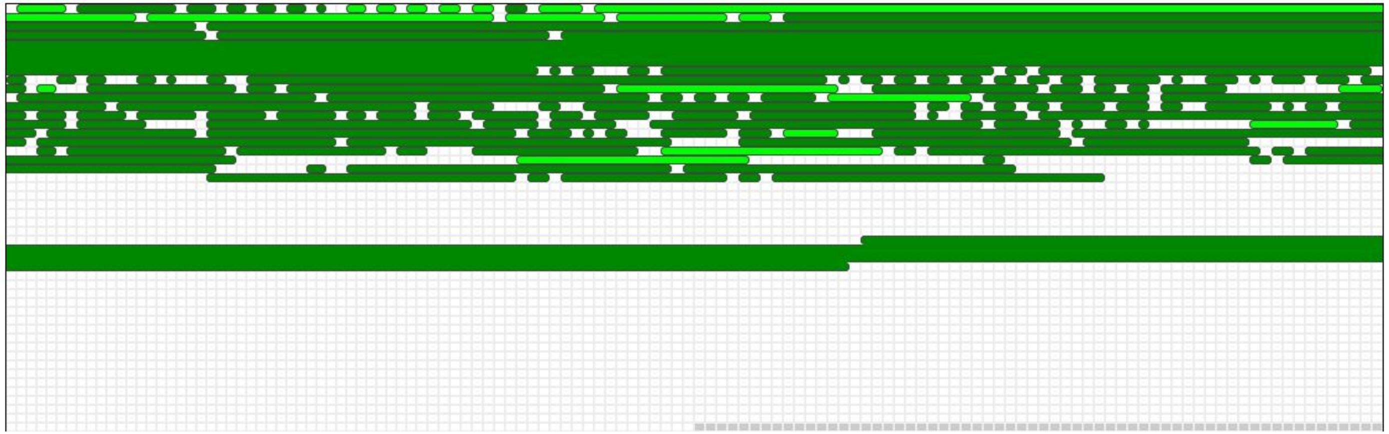
System Allocators and Allocation Process

- System allocator is an implementation of *malloc* and *free* functions that allow your program to allocate and deallocate memory on demand (not to be confused with STL allocators)
- Allocator internally asks for a large ***block*** of memory from the OS, and serves your program with smaller ***chunks*** when the program calls *malloc*. The program uses the returned chunk to keep the data.
- Allocation algorithms must be very fast, and finding the chunk of available memory is not an easy task



The problem of memory fragmentation

- As your program runs allocates and deallocates memory, it gets more and more difficult to find an empty slot of the right size - the problem of memory fragmentation
  - As a result, malloc and free take more and more time and your program is less and less responsive
- A serious problem for the long-running programs and systems - it causes slowdowns and allocation failures



Slow allocation and deallocation

- There are three reasons why allocation and deallocation is slow, here presented in the order of importance:
  - Your program is allocating and deallocating a lot of memory, especially small memory chunks
  - Memory fragmentation
  - Your program is using an inefficient implementation of malloc (new) and free (delete)
- Fixing any of the three above reasons will make allocation and deallocation in your program faster
- Often, decreasing the number of calls to malloc and free you will decrease memory fragmentation

Solution: don’t use vectors of pointers

- Some programming practices inevitably lead to many calls to malloc and free
- The worst offender: vector of pointers (std::vector<object*>, std::vector<std::unique_ptr<object>>, etc.)
  - A call to malloc for each element of the array
  - Additionally, bad performance due to poor data locality
    - Neighboring pointers don’t necessarily point to the neighboring objects in memory
- Alternatives:
  - Use separate vector per type: decreases the pressure on the memory allocator, additional benefit is excellent data locality. An example implementation in boost::polly_container or here: https://johnysswlab.com/the-true-price-of-virtual-functions-in-c/#multivector
  - Use std::vector<std::variant<...>>. This alternative is good if container needs to be sorted.
  - https://johnysswlab.com/process-polymorphic-classes-in-lightning-speed/#polymorphic
- Both approached decrease the number of calls to malloc and free. Also decrease memory fragmentation

Solution: use STL allocator for your data structure

- Fast access data structures based on trees and hash maps such as std::map, std::list, std::unordered_map allocate a lot of small memory chunks from the system allocator
  - This increases fragmentation and slows down the your program
- Alternative: a dedicated STL allocator for the data structure
  - STL allocator should allocate memory from a dedicated block of memory
  - Separating memory allocated by your data structure from other memory decreases the memory fragmentation and increases the speed of the system allocator
  - Implementation of your STL allocator can be very simple, and because of the simplicity it will be very fast
  - When the data structure is destroyed, the whole block of memory can be returned back to the operating system -> zero memory loss due to memory fragmentation
- The main reason why you would want to use a custom STL allocator is performance!

Custom allocator for STL containers: example

```
template <typename _Tp>
class zone_allocator
{
private:
   _Tp* my_memory;
   int free_block_index;
   static constexpr int mem_size = 1000*1024*1024;
public:
   zone_allocator()
   {
       my_memory = reinterpret_cast<_Tp*>(mmap(0,
mem_size, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0));
       free_block_index = 0;
   }
   ~zone_allocator()
   {
       munmap(my_memory, mem_size);
   }
   ....
```

```
    pointer allocate(size_type __n, const void* = 0)
    {
        pointer result = &my_memory[free_block_index];
        free_block_index += __n;
        return result;
    }

    void deallocate(pointer __p, size_type __n)
    {
       // We deallocate everything when destroyed
    }
  …
  };
```

```
std::map<int, my_class, std::less<int>,
zone_allocator<std::pair<const int, my_class>>>
```

Solution: Don’t allocate objects for communication

- A common pattern: passing messages between threads as objects
  - The sender thread allocates objects, the receiver thread deallocates them
  - A busy system can make a large amount of calls to the system allocator
  - Additionally, system allocators don’t do well with allocating memory in one thread and deallocating it in another
- Solution: don’t release memory back to the allocator, cache it

```
T* memory = allocator.get_memory_chunk();
new (memory) T(args…);
…
memory->~T();
allocator.release_memory_chunk();
```

- Decreases the memory fragmentation

Additional solutions to fight memory fragmentation

- Preallocate all the needed memory upfront
  - Often used in embedded and real-time systems, e.g. if a table can have at most 1000 entries, allocate a single table with 1000 entries (can be allocated in global memory)
- Restart the program
  - Long running programs are getting slower and slower due to memory fragmentation
  - A solution is to restart the program
  - Not always simple, sometimes you need to save the state of your program before restarting
- If a data structure is small most of the time, use *small buffer optimizations* to avoid small allocations on the heap -> example std::string
- Use special system allocators that promise low fragmentation
- Not every of the presented techniques is applicable everywhere

Off-the-shelf allocators

- There are a few good open source allocators that you can use in your projects
- No allocator is perfect for all applications, and you need to test them and verify them with your programs
- Things to keep in mind:
  - Allocation speed: speed for both malloc and free is important
  - Memory consumption: the percentage of memory that gets wasted in each block, due to allocation overhead or speed over consumption tradeoff: important for systems with little memory (e.g. embedded systems). Also, the amount of memory requested from the operating system but not used by the application
  - Memory fragmentation: important for long-running application
  - Data locality: if two consecutive calls to malloc return neighboring memory chunks, later, when traversing the memory chunks (in a vector, linked list, etc), the access speed will be much faster
- It’s not easy to measure all of the parameters by just measuring the runtime!

Off-the-shelf allocators (2)

- Standard C library provides implementations of *malloc* and *free*. These are most commonly available on Linux
- Other allocators on Linux:
    - tcmalloc (Google)
    - jemalloc (Facebook)
    - mimalloc (Microsoft)
    - hoard allocator
    - ptmalloc
    - dlmalloc
- Installable from the repositories.
- Each of them makes certain tradeoffs for speed and memory consumptions.
- There is no good or bad, try them, measure speed and consumption and see which one suits you best

Allocator test

- Allocators come as libraries, that you can either link against, or replace them in runtime (on Linux only)
- On Linux, you can use *LD_PRELOAD* environment variable to overwrite the default allocator

```
$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libtcmalloc_minimal.so.4
./my_program
```

- We tested the performance of a few allocators (GNU, Google’s, Facebook’s, Microsoft’s).

Allocator test - results

- We used *kaldi* speech recognition software which puts a large pressure on the system allocator: 7.7% of time in *malloc* and 1.54% of time in free
- Single threaded program that uses linked lists to store data

Allocator	Runtime	Memory usage
Builtin	27 s	288 MB
TCMalloc	24.7 s	289 MB
JEMalloc	24.7 s	288 MB
MiMalloc	24 s	292 MB

Performance of memory access

Memory Access Performance

- If your program allocates and deallocates a lot of memory, the performance of the allocator will be important for the overall speed
- But, your program's performance will depend on the way you access your memory, more specifically:
  - How is your data laid out in memory?
    - Much better to lay out data sequentially than to throw it around the memory
  - What is the access pattern to your data?
    - Sequential access pattern is much better than random access pattern or strided access pattern
- Simple abstraction of malloc/free that doesn't take into account the underlying hardware doesn't cut it for high-performance software
  - Highest performance can be achieved only if the allocator abstraction is broken, i.e. if the algorithm is aware of how the allocator works in order to allocate memory optimally

Why does the program’s speed depend on the memory access pattern?

- Memory speed is a bottleneck on modern systems
    - CPU typically spends around 200-300 cycles waiting for the value it needs to be fetched from the memory to a internal CPU register
        - In that time it can do 200-300 simple instructions
- To remedy this, the CPU designers introduced a small on-CPU memory called *cache memory*
    - This is a fast memory (3-15 cycles per access) where the CPU keeps data it is currently using called *dataset*

Cache memories - data locality

- Every access to the main memory begins by checking if the data is already present in the cache memory
- If the data is present in the cache memory, it can be loaded very quickly, otherwise, the CPU needs to wait for the data to be fetched from the main memory
- When the CPU doesn’t access data for some time, the modified data is returned back from the cache to the main memory to make space for new data. The process is called *eviction*

Cache memories - data locality (example)

- We are performing 64M searches on a hash set (std::unordered_set).
- There are three hash map sizes: small (32), medium (1M) and large (64 M).

	Small hash set (32)	Medium hash set (1M)	Large hash set (64M)
Runtime (in s)	1.5 seconds	2.8 seconds	5.9 seconds

- The larger the hash map, the smaller the chance that we will access the same data again before it gets evicted
- Applies to all random access data structures: arrays with random access, hash maps and trees

Example taken from: https://github.com/ibogosavljevic/johnysswlab/tree/master/2020-11-cachebatchprocessing

Cache memories - prefetcher

- If your program has a predictable access pattern (typically, iterating linearly over a vector of objects), the cache memory prefetcher can figure that out and load the data from the memory before its needed by the CPU
- Programs that access memory in the predictable fashion will see major improvements in speed

Cache memories - prefetcher (example)

- Summing array containing 100M integers:

	sequential: i = 0, 1, 2, 3, ...	strided: i = 0, 16, 32, 48, ...	random: i = random
Runtime	180 ms	740 ms	6600 ms

- Prefetcher works with sequential and strided memory accesses
- Prefetcher is powerless with random memory accesses

Source: https://github.com/ibogosavljevic/johnysswlab/blob/master/talks/dynamic-memory/access_pattern_test.cpp

Cache memories - cache line

- Cache memories are divided into cache lines (typically 64 bytes in modern systems)
- Each line in cache corresponds to a block of the same size in memory
- Access to one byte inside a cache line means that the whole line will be fetched to the line
- Access to any other byte in the same cache line is very fast
- To benefit from the cache line organization:
    - Organize their data so that the data that is accessed together is close to one another in memory
    - Use all the data in the cache line while processing data

Cache memories - cache line (example)

- Summing array containing 100M integers:

Stride	Percentage of data actually read from the cache line	Runtime
stride 1: 0, 1, 2, 3	100%	180 ms
stride 2: 0, 2, 4, 8	50%	210 ms
stride 3: 0, 3, 6, 9	33 %	240 ms
stride 4: 0, 4, 8, 12	25 %	270 ms

"It is a sin to load the data into the cache line and then not use it"

Source: https://github.com/ibogosavljevic/johnysswlab/blob/master/talks/dynamic-memory/access_pattern_test.cpp

Summary: Memory Access Performance

- CPU always works with simple types: char, int, double, float, etc.
- From the performance point of view, dependence between the memory access pattern and the access speed looks like this:
  - Sequential access: you are accessing neighboring simple types - best performance
    - vector<int> a; int sum = 0; for (i = 0; i < a.size(); i++) { sum += a[i]; }
  - Strided access: you are accessing simple type in a vector of class instances - bad for performance, the bigger the class, the worse the performance
    - vector<rectangle> a; int sum = 0; for (i = 0; i < a.size(); i++) { sum += a[i].visible; }
  - Random access: you are randomly accessing objects in memory (std::set, std::map, std::list), or accessing an object through a pointer
    - set<rectangle> a; int sum = 0; for (auto& r: a) { sum += r.visible; }
    - class car { driver* m_driver; }; if (my_car.m_driver->experience() > 5) { … }

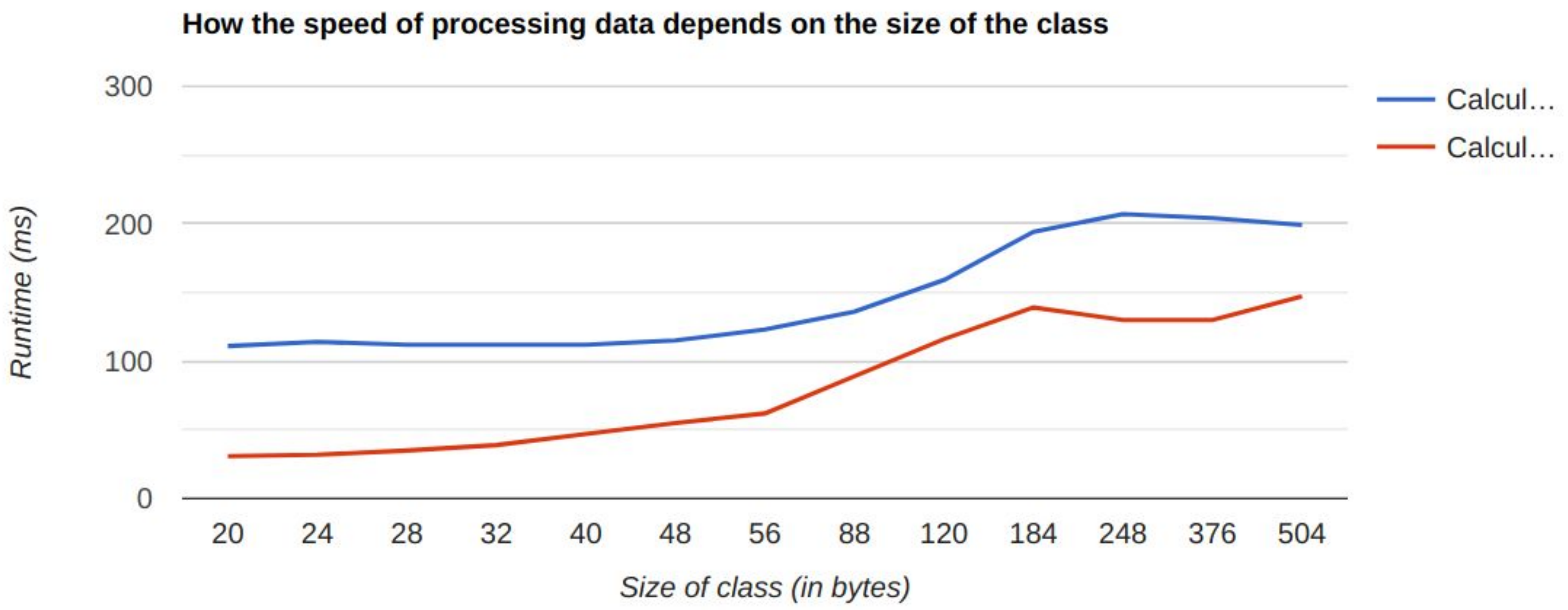
Experiment with class size and member layout

```
template <int pad1_size, int pad2_size>
class rectangle {
  private:
   bool m_visible;
   int m_padding1[pad1_size];
   point m_p1;
   point m_p2;
   int m_padding2[pad2_size];
};
```

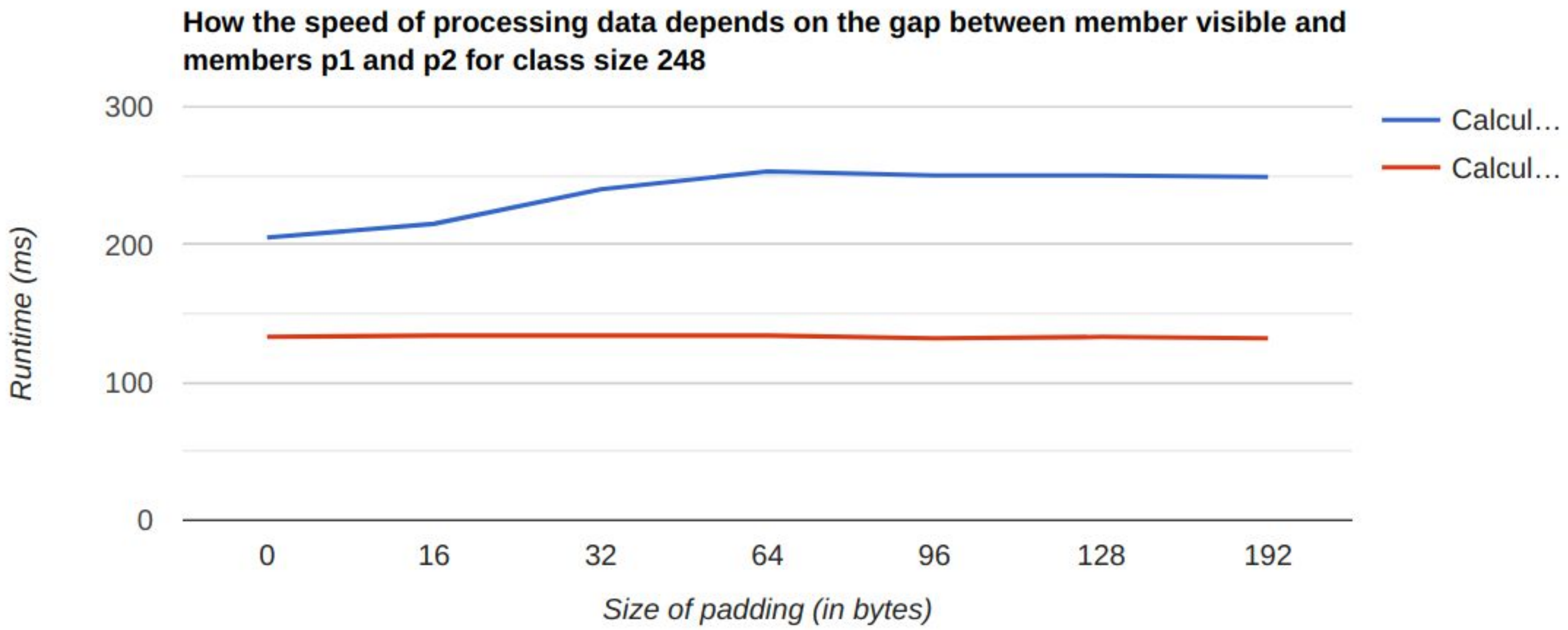
```
template <typename R>
int calculate_surface_visible(std::vector<R>&
rectangles) {
   int sum = 0;
   for (int i = 0; i < rectangles.size(); i++)
{
       if (rectangles[i].is_visible()) {
           sum += rectangles[i].surface();
       }
   }
   return sum;
}
```

Source: https://johnysswlab.com/2-minute-read-class-size-member-layout-and-speed/

Experiment with class size and member layout



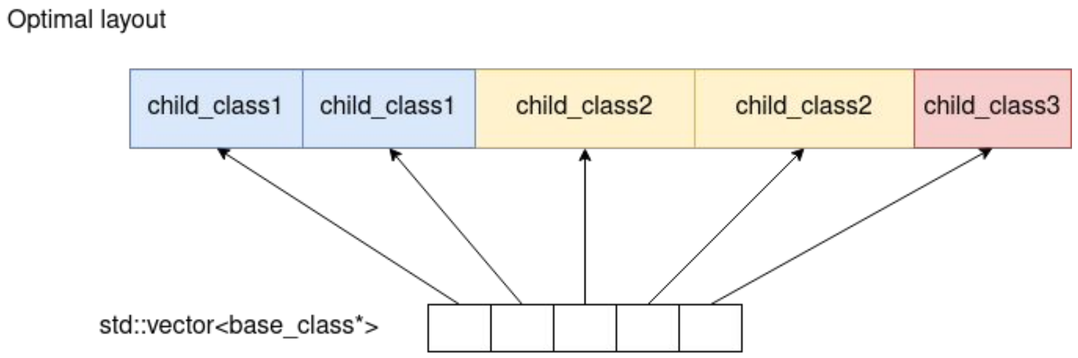
Experiment with class size and member layout

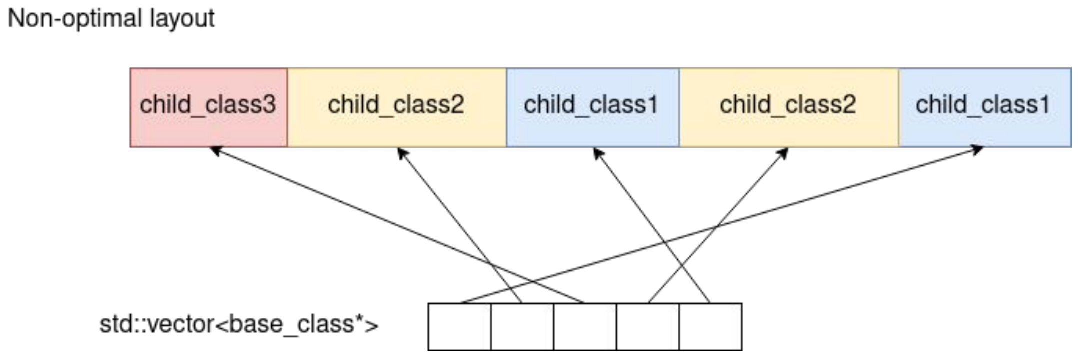


Principles of cache-aware software design (1/4)

- Make data accesses predictable
  - Iterating over a vector of objects is almost always faster than iterating over a vector of pointers (we already talked about the alternatives for the vector of pointers)
  - Use fast access data structures sparingly: CPU needs to “chase pointers” for trees (std::set and std::set) and hash maps (std::unordered_set and std::unordered_map). The memory access pattern is unpredictable and the hardware prefetcher is powerless.
  - Classes that contain pointers to other heap-allocated classes will suffer from low performance when the pointer to the other class is dereferenced
  - Don’t use linked lists (std::list). Instead use some of the array based alternatives (e.g. colony, bucket array or a gap buffer)

Array of pointers





Array of values vs array of pointers

- Arrays of values are much better for performance compared to array of pointers
  - All memory allocated in a single block
  - Sequential access to objects translates to sequential access to memory addresses
  - No calls to malloc/free
  - No virtual dispatching mechanism to slow things down
  - Enables small function inlining because type is known at compile time
  - Downside: no polymorphism
- For speed, prefer arrays of values.
- It is possible to implement arrays of values with polymorphism. Check out the article: https://johnysswlab.com/process-polymorphic-classes-in-lightning-speed/

Array of pointer performance

- Performance of a vector with 20 million objects

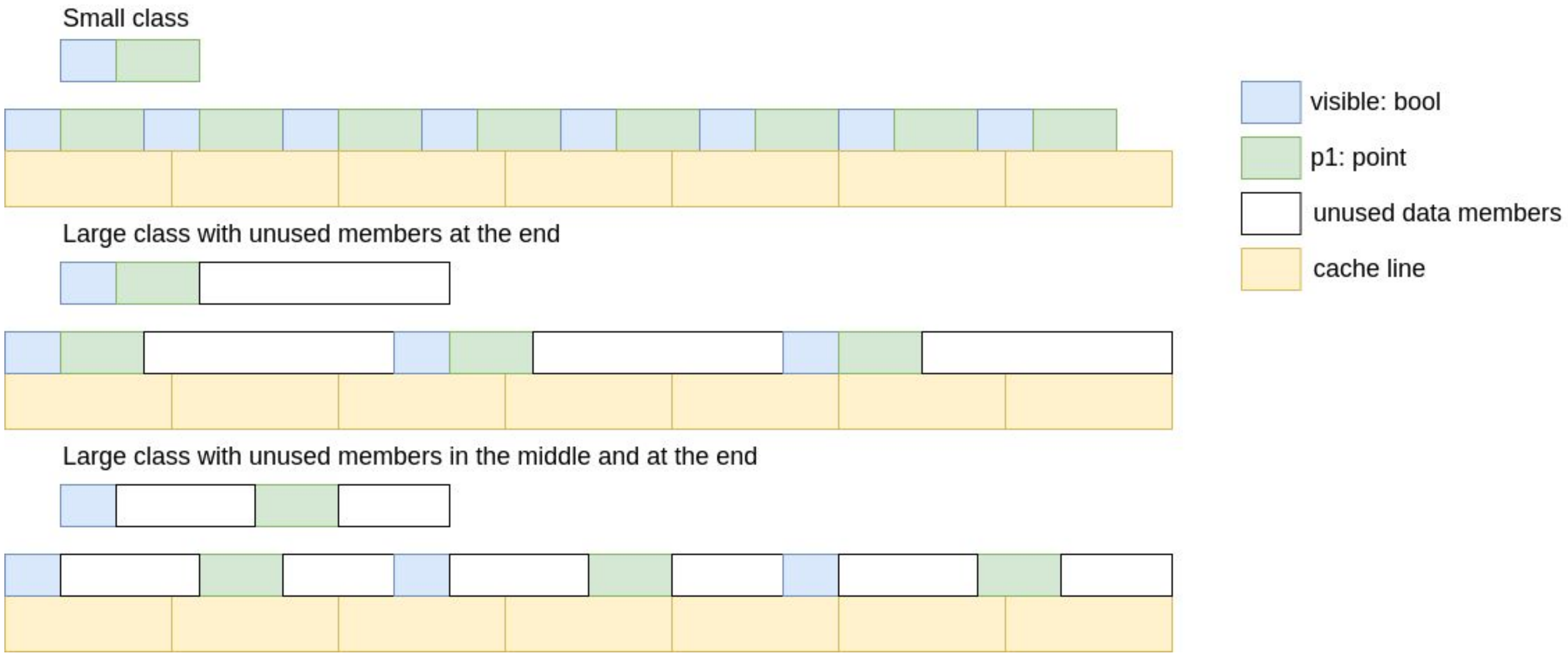
	Perfectly optimal layout	Perfectly non-optimal layout
Small virtual function	384 ms	5067 ms
Large virtual function (with many computations)	2660 ms	16905 ms

Source: https://johnysswlab.com/process-polymorphic-classes-in-lightning-speed/

Principles of cache-aware software design (2/4)

- The whole block of 64 bytes is loaded into memory, don't let data in a cache line go wasted
  - Keep your classes small
    - Data members that are used together should be kept together, everything else should be part of another class
      - You want to load only data into data cache that you actually use
      - Data loaded into cache but not used is wasted performance opportunity
  - Declare data members you access together close to each other
    - This increases the likelihood that the data members will be in the same cache line
  - Keep your small classes in a dedicated vector; avoid processing small classes as part of a larger class
    - Process small classes by iterating over their dedicated vector
    - Don't use vector of pointers, use vector of values
- Further read: https://johnysswlab.com/2-minute-read-class-size-member-layout-and-speed/

Small vs Large classes memory layout


Small class
Large class with unused members at the end
Large class with unused members in the middle and at the end
visible: bool
p1: point
unused data members
cache line

Class Size, Data Layout and Performance

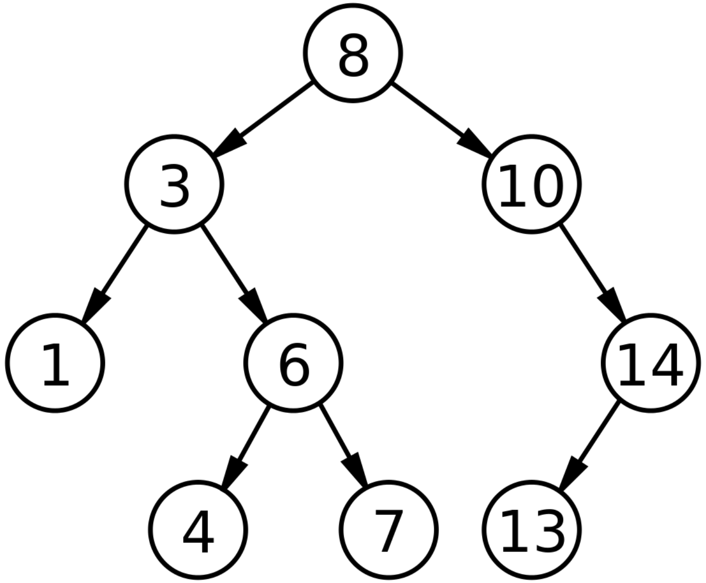
- Object Oriented Paradigm in C++ can be inefficient from the performance perspective
  - Containers of pointers
  - Containers containing objects of different types
  - Large classes
  - Classes having members that point to other heap allocated classes
- In game development, they use a paradigm called Entity–component–system (ECS)
  - Get rids of large classes
  - Get rids of inheritance
  - No inheritance, instead an entity which consists of components
  - Components are processed independent of the entity they belong
  - Entity can change its “type” at runtime
- I don’t think the world is moving towards ECS, however, there are some principles in ECS that can be useful in all programs when the speed is important

Principles of cache-aware software design (3/4)

- The whole block of 64 bytes is loaded into memory, don't let data in a cache line go wasted
  - Prefer n-ary tree implementation to binary tree, as they use cache lines more effectively
    - Added bonus: n-ary trees are shallower than binary trees, shorter path between the root and the leaf node
- Don't store pointers in a hash map, store whole objects
  - Having pointers in a hash map adds another cache miss due to indirect access required by a pointer
- Use cache friendly hash maps based on open addressing scheme
  - In case of collision, they perform better in terms of data cache misses
  - Beware that open addressing hash maps have downsides when the load factor is high

Binary Tree Example

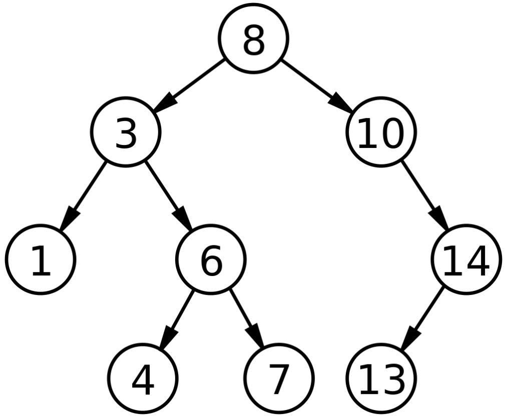
- Binary Tree - a data structure used for fast lookup - to check if the value is present in the binary tree, insert the value or remove it



Binary Tree Example - Memory Layout

- Each node in the binary tree is represented with a node

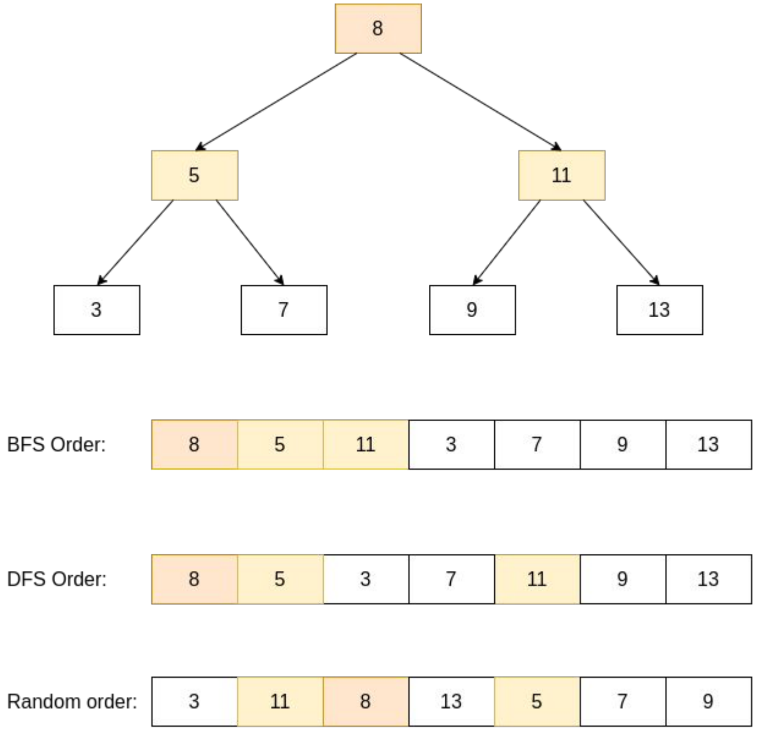
```
template <typename T>
struct node {
    T value;
    node* left;
    node* right;
};
```



- Memory is one-dimensional, whereas binary layout structure is two-dimensional.
- Question: how to optimally represent this memory structure in memory?

Binary Tree Example - Memory Layout

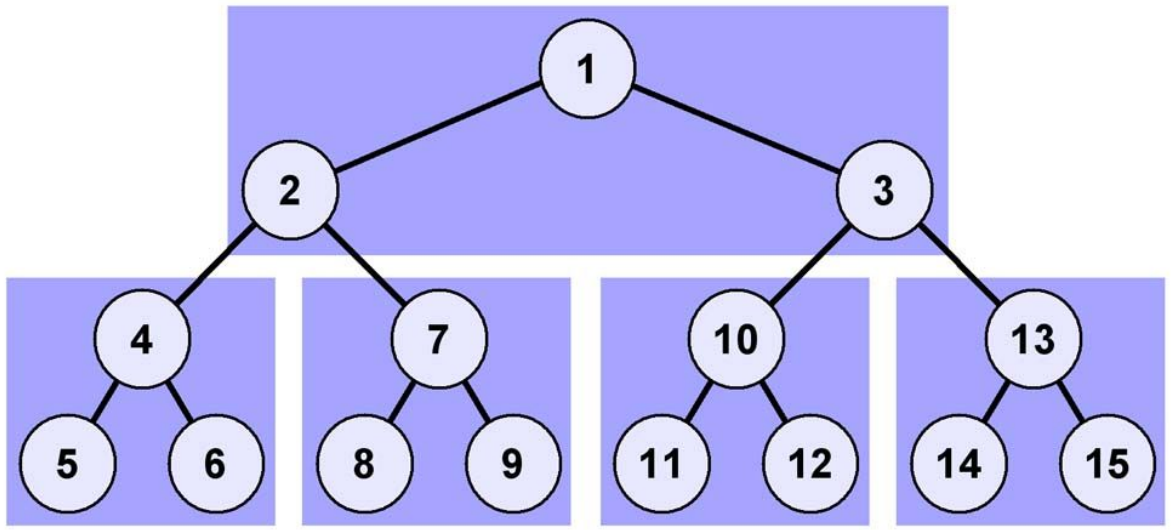
- Three memory layouts for the same structure
  - BFS (breadth-first search) layout: we put first nodes on the first level, then nodes on the second level, etc.
  - DFS (depth-first search) layout: we visit the current node. If it has a left subtree, we go and visit it. If it has a right subtree, we visit the right subtree
  - Random order: we don't care about the memory layout. We just ask for memory chunks from the allocator, and the allocator simply returns them
- Poll: which is the most optimal layout to check if the given list of values is present in the tree?



8
5
11
3
7
9
13
BFS Order: 8 5 11 3 7 9 13
DFS Order: 8 5 3 7 11 9 13
Random order: 3 11 8 13 5 7 9

Binary Tree Example - Memory Layout

- Van Emde Boas layout



Binary Tree Example - Lookup performance

- A perfect binary tree created from a sorted array. Perfectly balanced. Memory allocated by calling malloc on the allocator. Three kinds of allocators.

	BFS layout	DFS layout	Van Emde Boas layout
std::allocator Runtime	12.8 s	8.3 s	7.3 s
Optimal Allocator Runtime	11.2 s	7.2 s	6.4 s
Non-optimal Allocator Runtime	16.7 s	14.3 s	14.4 s

Source: https://johnysswlab.com/the-price-of-dynamic-memory-memory-access/

Binary Tree Example - Memory Layout

- We gain performance if:
  - We allocate a dedicated block of memory for the data structure
    - The related data is kept in one place - better data cache hit rate
    - We can achieve this only with a custom allocator
    - Added benefit: when the data structure is destroyed, we can release the whole block to the operating system
  - We try to keep the block of memory as compact as possible
    - Easy to do with custom allocator
    - Increases data cache hit rate
  - We take advantage of cache line organization
    - If two nodes are adjacent in the tree and they are adjacent in memory, there is a high probability that they will share the same cache line
    - If they share the same cache line, we get the access with no cache misses

Binary Tree Example - Memory Layout

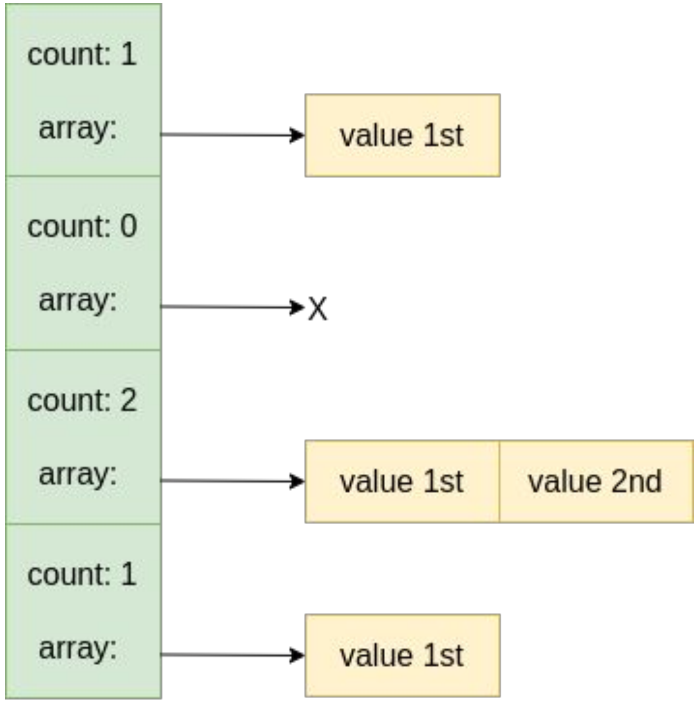
- We gain performance if:
  - We keep the *struct node* as compact as possible
    - This increases the likelihood that two or more nodes share the same cache line
    - On 64 bit system, only 48 bits of a pointer are actually used. We can combine two pointers to decrease the size of *struct node*
    - Recompiling for 32 bit system can improve speed due to smaller pointers
    - Idea: use std::vector as an allocator for the tree

Binary Tree Modification

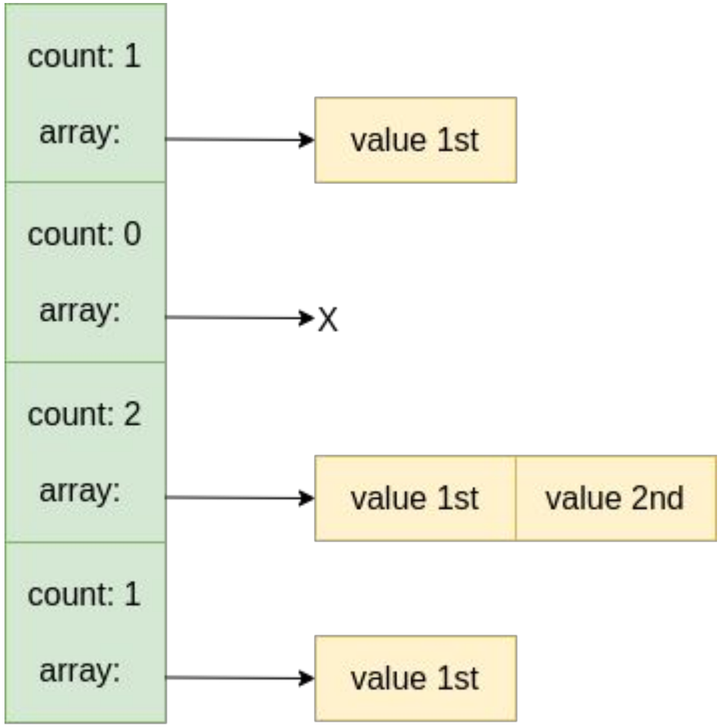
- Adding and removing nodes slowly makes the memory layout less and less optimal
- After some time, the access becomes slower and slower
- Solutions:
  - Recreate the data structure which is optimal again or
  - Perform *defragmentation* of the existing data structure or
  - Don’t delete the nodes. Keep them around for some time so they can be reused if opportunity arises
  - Both of this takes time, but can speed up your long-running program

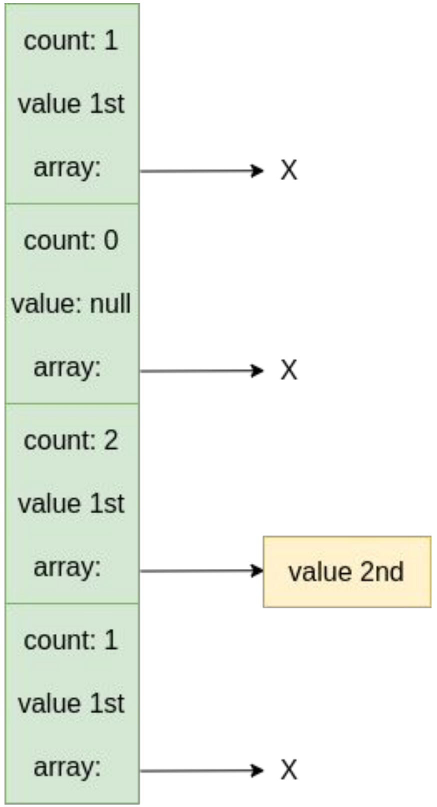
Hash Map Example

- An example hash map uses an array to store its values
- Inside each element of the array, there are:
  - A counter that counts the number of elements in the array
  - A pointer to the the array the holds the values
- Typically, entry will be empty or have one value. Two or more values are called collisions and hash maps avoid them by growing
- What is problem with this approach?



Optimized Hash Map





Optimized Hash Map - lookup performance

Hash map implementation	Large load (64M entries, 1 iteration)	Medium load (1M entries, 64 iterations)	Small load (32 entries, 2M iterations)
std::unordered_set	5853 ms	2849 ms	1435 ms
Simple hash map	5838 ms	4161 ms	1824 ms
Optimized hash map	3785 ms	3184 ms	1427 ms

Principles of cache-aware software design (4/4)

- Don’t let the data you will reuse in the future get evicted from the cache
  - Don’t iterate the same container two times (e.g. first time to find the maximum and second time to find the minimum)
  - If your program is performing many random accesses, to increase the likelihood that the same data is not evicted from the data cache, you should try do decrease the size of your data structure:
    - Move rarely used members out of the hot class
    - Recompile for 32 bit system (this makes the pointers smaller, so the data structure which internally uses pointers get smaller)
    - Rewrite the code not to use pointers, but to use offset from a fixed array.

Reiterating the same collection two times

- Finding minimum and maximum in the vector (class size 16 bytes), 100M elements in the vector

	Finding min and max in separate loops	Finding min and max in the same loop
Runtime	368 ms	207 ms

Recompiling binary tree example for 32 bit system

	BFS layout	DFS layout	Van Emde Boas layout
Optimal Allocator Runtime (64-bit)	11.2 s	7.2 s	6.4 s
Optimal Allocator Runtime (32-bit)	9.8 s	6.5 s	5.9 s

When not to optimize?

- Some of the techniques presented here will benefit only the cases with large data sets
  - Binary tree optimizations and hash map optimizations only make sense if the data set is large.
  - It will probably not pay off for really small vectors as well
- Small data set is data set that fits L1 cache
  - On most CPUs the size of the data set is 16 kB - 32 kB
- Large data set is data set that doesn’t fit the Last Level Cache
  - On most CPUs the size of the LL cache is few megabytes
- Data structures with a short life span also don’t benefit from optimizations a lot
  - Creating a memory optimal data structure takes more time than an average data structure
  - If the data structure is used a few times and then destroyed, the overhead of creation will dominate the runtime

Final Words

- Most of today's software is limited in speed due to limit in memory bandwidth, not limit in computational power.
- Reckless use of computers resources can lead to software that is much slower than it needs to be
- Careful design can help mitigate some of those problem if the performance is important
    - The developers need to be aware of the memory hierarchy in the modern computers and how to take the full advantage of it
- For performance enthusiasts: Electronic Arts (the game developer) has its own implementation of STL with focus on performance (as opposed to simplicity)
    - Many great ideas on how to do optimizations for performance sensitive applications
    - Google "EA STL" for more information

The End

- Most material taken from articles from the blog Johny’s Software Lab:
  - *The price of dynamic memory: Allocation*
  - *The price of dynamic memory: Memory Access*
  - *Process polymorphic classes in lightning speed*
  - *Use explicit data prefetching to faster process your data structure*

Thank you for your attention!

Ivica Bogosavljević for Johny’s Software Lab
https://johnysswlab.com