Johny’s Software Lab

The price of dynamic memory in C and C++

How much does using dynamic memory actually costs in terms of performance?

https://johnysswlab.com, @johnysswlab ivica@johnysswlab.com

Introduction

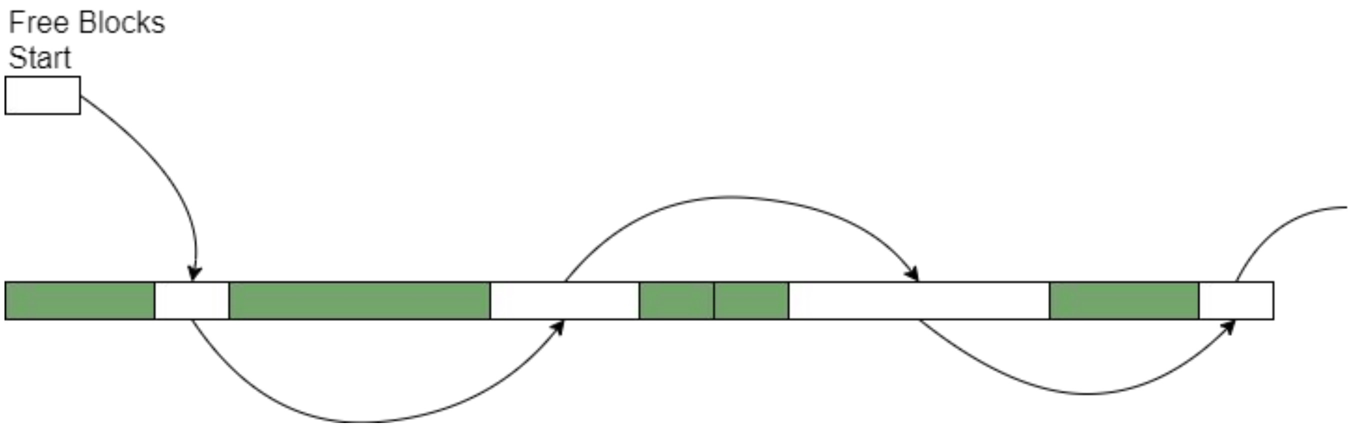
- Two types of programs when it comes to memory usage
  - All allocations are a few large blocks of memory
    - Typically (but not always) these are arrays that hold data, and the processing is done either sequentially or in random access mode
  - The program allocates many blocks during the program lifetime
    - Programs that keep information in random access data structures (trees or hash maps)
    - Any program that allocates many instances of a single class
- 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

Allocators and allocation process

- Allocator is an implementation of *malloc* and *free* functions that allow your program to allocate and deallocate memory on demand
- Allocator internally ask 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, but finding the chunk of available memory is not an easy task

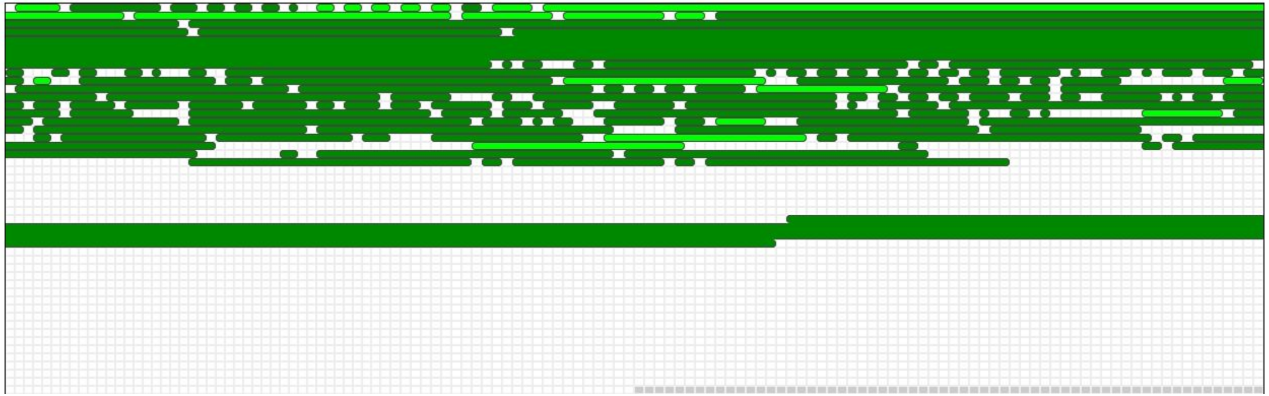


Allocators

- In the next few slides we will talk about how are allocators implemented and what challenges they face:
  - This will help us understand why they are slow, and strategies you can use to mitigate this
  - It will help you pick one up off the shelf, in case you opt for a off-the-shelf allocator
  - Occasionally you might want to implement your own allocator for a specific application, and this will help you understand the problems you will face

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 allocation failures and slowdowns



Memory fragmentation

- How to tackle the memory fragmentation?
  - Occasionally restart your program
  - Preallocate all the needed memory at the program beginning
  - Cache memory chunks
  - Use special memory allocators that promise low fragmentation
- Not every technique is applicable everywhere

Allocators and thread synchronization

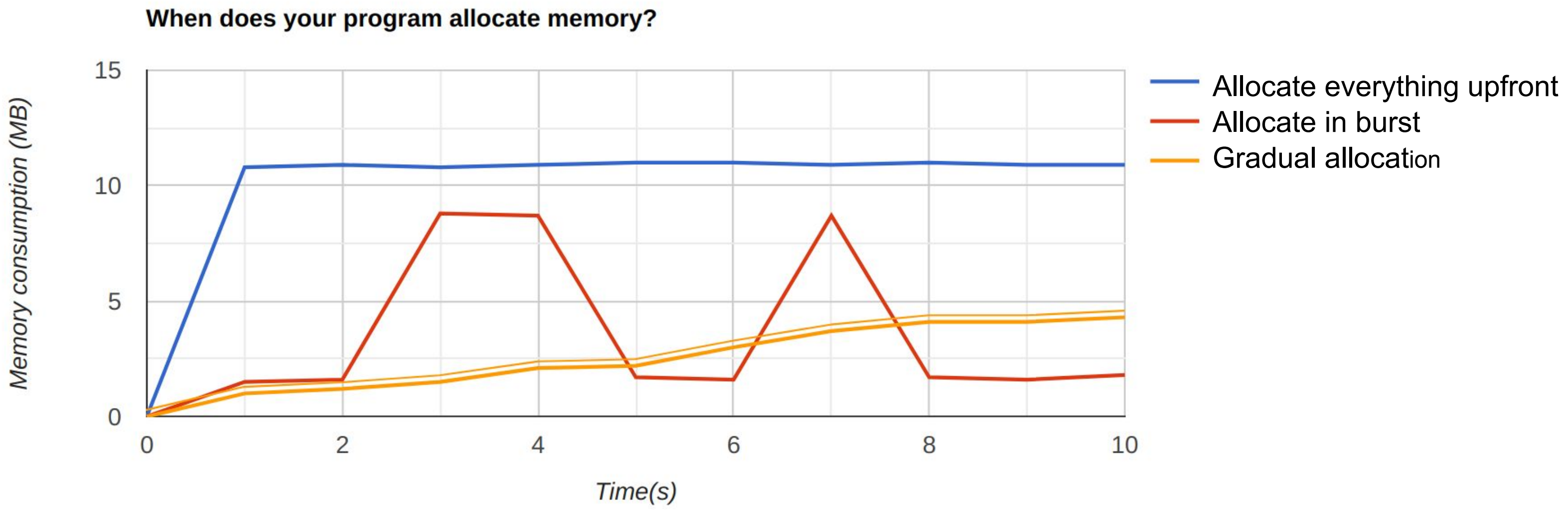
- If the allocator internally uses one block of memory to allocate memory for several threads, it needs to protect the critical section with a mutex
- This can slow down the allocator
- Many allocators solve this problem by using per thread memory blocks
- Increases memory consumption, but removes the synchronization penalty
- Problem: what happens if the program allocates a memory block in one thread, but releases it in another thread?

Memory allocation pattern

- If your program is slow because it allocates and deallocates a lot of chunks, you need to understand your program’s memory memory allocation pattern
- Memory allocation pattern can influence the behavior of the allocator
- Allocation pattern will tell you what kind of optimizations you can use

When does your program allocate memory?

- Upfront: preallocate everything
- Allocation in bursts: introduce caching; invest in a good allocator. Use custom allocator for STL containers
- Gradual allocation: Use a good allocator.



Custom allocators for STL containers

- C++11 STL containers (vectors, maps, unordered_maps) allow programmers to provide a custom allocator as a template parameter
- An excellent choice when a data structure (map or set) performs a lot of small allocations
- Using a custom allocator, the programmer can control the allocation
- Allocator implements *allocate* and *deallocate* methods that the data structure uses to get and release memory

Custom allocator for STL containers: example

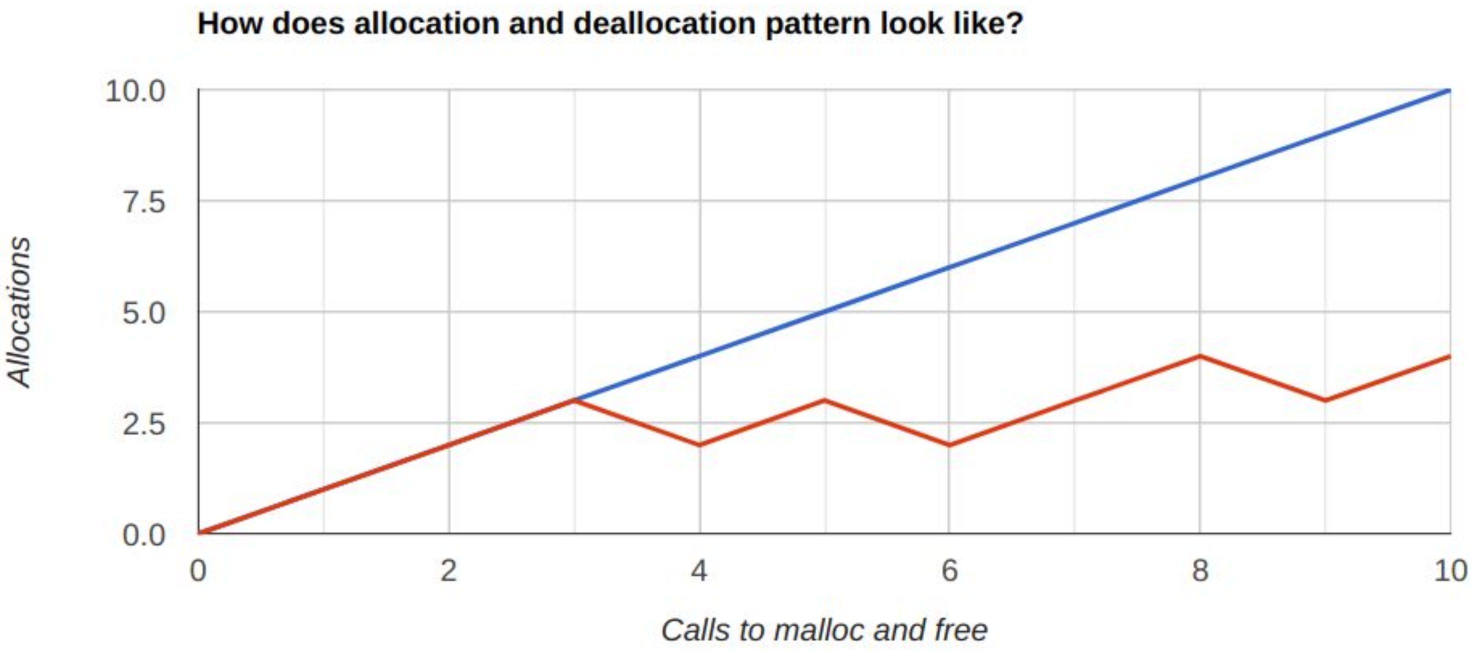
```
template <typename _Tp>
class zone_allocator
{
private:
    _Tp* my_memory;
    int free_block_index;
    static constexpr int mem_size = 1000*1024*1024;
public:
    zone_allocator()
    {
        my_memory = reinterpret_cast<_Tp*>(mmap(0,
mem_size, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS,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

How does allocation and deallocation pattern look like?



Memory chunk caching

- If your program is performing many allocations and deallocations on the objects of the same size, your program's performance can suffer from memory fragmentation
- One of the ways to mitigate this is to cache memory chunks: instead of returning it with *free*, we are keeping it for fast access later

Memory chunk caching

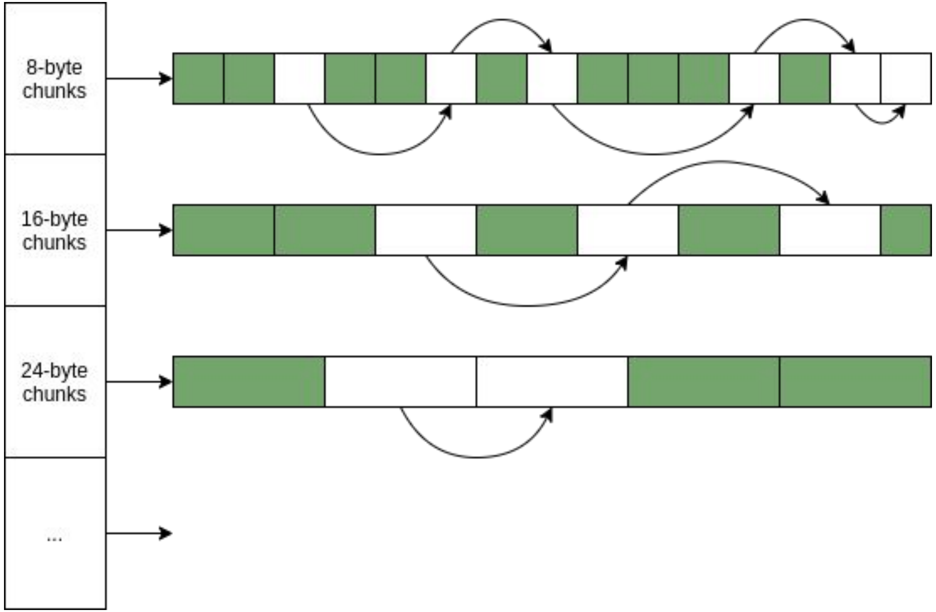
```
class chunk {
private:
   static int chunk_count = 0;
   static const int max_chunk_count = 30;
   static chunk* first_chunk = nullptr;
   chunk* next_chunk;
  ....
public:
   void * operator new(size_t size) {
       if (first_chunk) {
           void* res = first_chunk;
           first_chunk = first_chunk->next_chunk;
           chunk_count--;
           return res;
       } else {
           return malloc(size);
       }
   }
```

```
   void operator delete(void * p) {
       if (chunk_count <  max_chunk_count) {
           chunk* c = reinterpret_cast<chunk>(p);
           c->next_chunk = first_chunk;
           first_chunk = c;
           Chunk_count++;
       } else {
           free(p);
       }
   }
};
```

Other things to keep in mind

- Does your program allocate memory in many small chunks or few larger chunks?
- Is memory allocated and deallocated in the same order?
- Does your program allocates memory in one thread and then deallocates it in another?
  - Allocating chunks in one thread and releasing them in another typically lowers performance?
- Is your program a long running one?
  - Consider implementing the ability to save your program’s state to a file, restart it, and load its state from the file
  - Use allocation pools: each chunk size is allocated from a dedicated block -> works very well with systems with large virtual memory space (64 bit architectures)

Per chunk size allocation pools



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)
  - Memory fragmentation: important for long-running application
  - Cache locality: if returned chunk is in the data cache, first access to it will be fast

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

Hands-on

- 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
```

- Let's test the performance of a few allocators. Example in the terminal

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?
  - What is the access pattern to your data?
- 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

Cache memories

- 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*
  - Every access to in-memory data begins by the data being fetched from the main memory to the cache memory
  - When the data in the cache memory is not used for some time, it is automatically removed from the cache back to the main memory in a process called *eviction* to make space for new data
  - The CPU has a component called *data prefetcher*: if the CPU can figure out memory access pattern, it can prefetch data from the main memory into the cache memory before the data itself is needed

Cache memories - analogy with books

- On your desk you have place for 8 books, therefore you will keep 8 books that you actually need, and keep the other books in the library
- If you need a new book, you will return the least recently used back to the library
- If your library is sorted, for example by author, and you need to write about authors in alphabetical order, you can *prefetch* the book you will need in advance

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 cache line is very fast
- Programs that organize their data so that the data that is accessed together is close to one another in memory benefit from performance improvements

Cache memories - cache lines

- From the performance point of view, which implementation is better?

```
template <int count = 10>
class my_vector {
    int used;
    int values[count];
    int sum();
};
```

```
template <int count = 10>
class my_vector {
    int values[count];
    int used;
    int sum();
};
```

```
int my_vector::sum() {
    int result = 0;
    for (int i = 0; i < used; i++) {
        result += values[i];
    }
    return result;
}
```

https://johnysswlab.com, @johnysswlab ivica@johnysswlab.com

Cache memories - prefetching

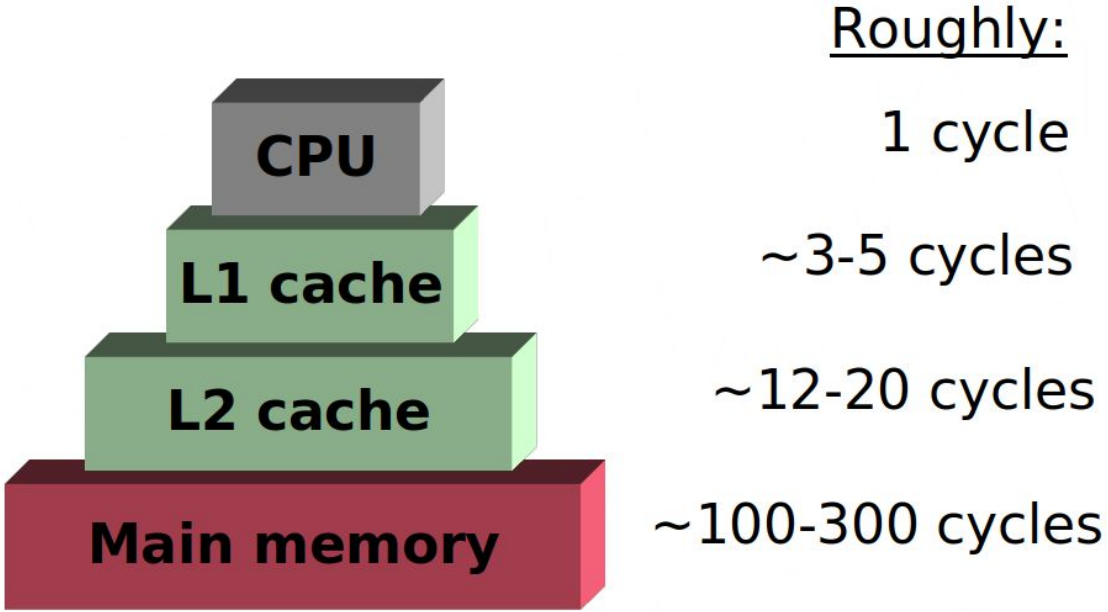
- If the hardware can figure out the memory access pattern, it will prefetch data from main memory before the CPU even needs it
- If we are accessing memory sequentially - one by one - the prefetcher will figure this out
- It works with forward and backward, where the stride is constant (1 or more)
  - Yet, the smaller the stride, the better the performance.
  - Best performance with stride 1. Why?
- Example in the command line

Cache memory types

- There are several cache memories available
  - Data cache memory - data needed by the instructions is kept in data cache memory
    - This is the most important cache memory, which when not used correctly, causes slow downs in your program
    - Most of the performance work is done here
  - Instruction cache memory - instructions themselves are kept in this cache
    - Normally, the compiler that generates instructions is responsible for generating the optimal instruction flow
    - Occasionally requires tweaking by the developer
  - TLB cache memory - used for translating virtual addresses to physical addresses
    - Normally not the bottleneck, except for very large data sets that are accessed randomly (hash maps or trees)
    - Operating system offer *large/huge* virtual pages that mitigate the TLB cache misses, but the OS and the application needs to be configured to use them

Cache Memory Levels

- Caches are typically divided into levels, where lower levels are faster and smaller in capacity

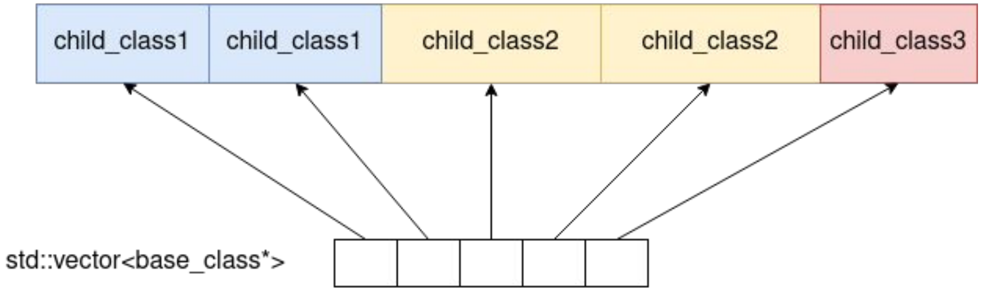


Data cache and arrays

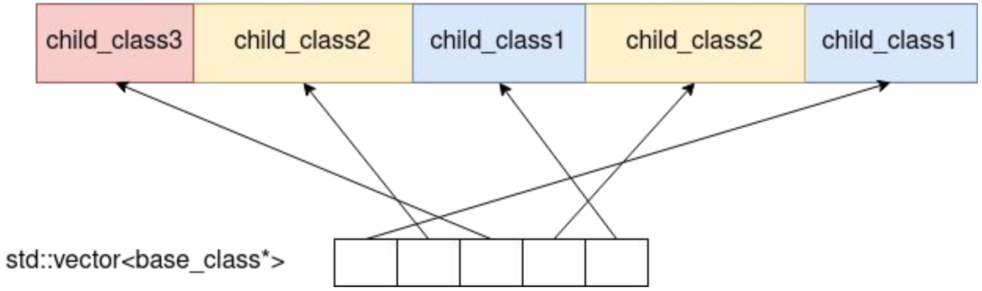
- From the HW perspective, arrays (vectors) with sequential access are the best way to process data
  - Dereferencing a pointer often creates memory cache miss and CPU stall (valid for linked lists, trees, hash maps) etc.
- In C++, polymorphism is achieved by using pointers
  - std::vector<base_class*>
- This can be very inefficient from the performance point of view
  - If the dataset is large (more than a megabyte), it will not fit into the data cache
  - We can expect a huge slow-down in those cases, since memory pointed by the pointers doesn't necessarily has to be consecutive in memory - hardware prefetching will not work
  - For small data structures that fit nicely into the cache, you will see very little performance gain, or even performance regression

Array of pointers

Optimal layout



Non-optimal layout



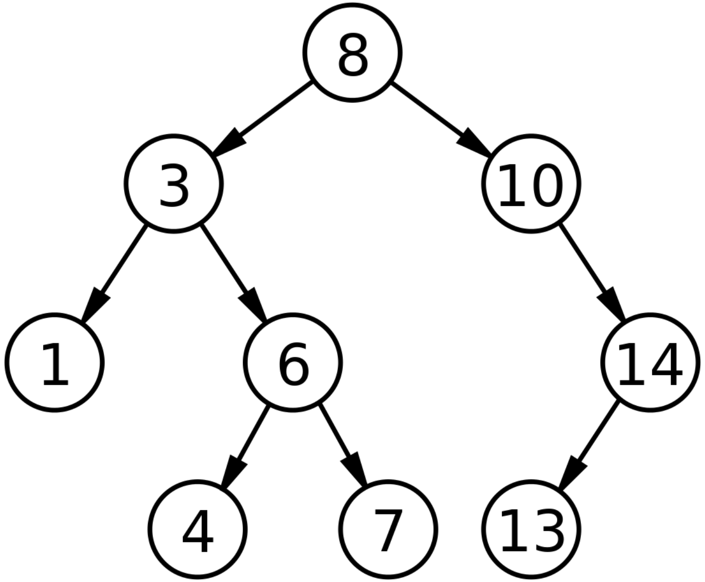
https://johnysswlab.com, @johnysswlab ivica@johnysswlab.com

Array of values vs array of pointers

- Taken from article *Process polymorphic arrays in lightning speed*
- 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 team
  - Downside: no polymorphism
- For speed, prefer arrays of values.
- It is possible to implement arrays of values with polymorphism. Check out the article
- Example in the command line

Binary Tree Example

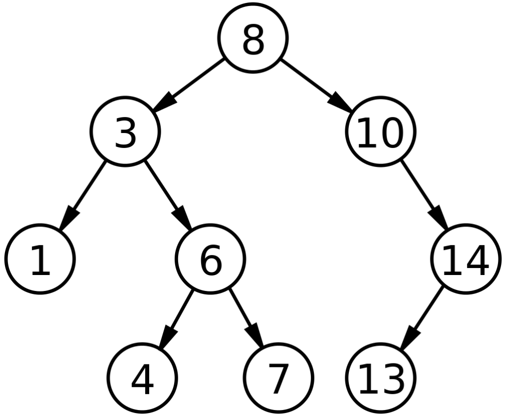
- Binary Tree - a data structure used for fast lookup - to check if the value is present in the binary tree, insert the value or remove it



Binary Tree Example - Memory Layout

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

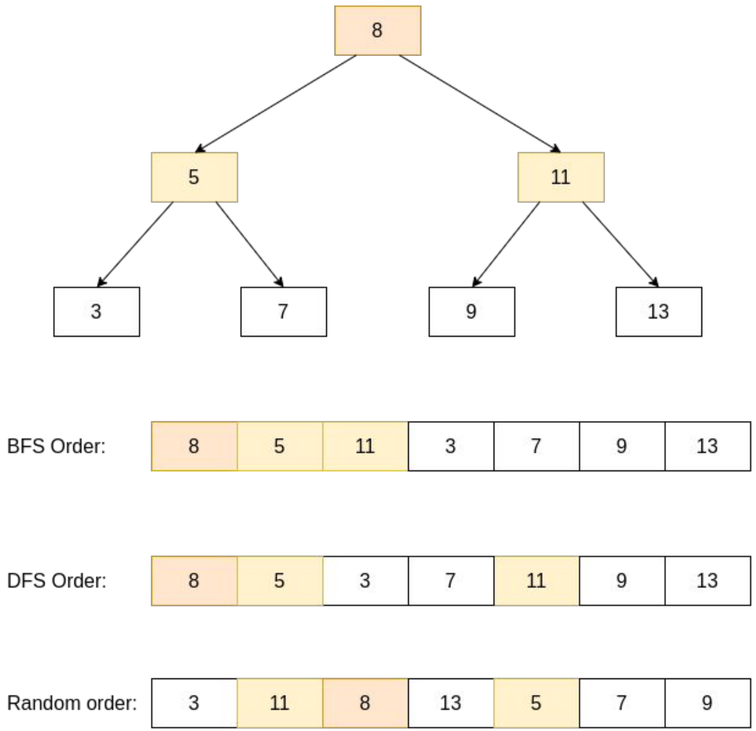
```
template <typename T>
struct node {
    T value;
    node* left;
    node* right;
};
```



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

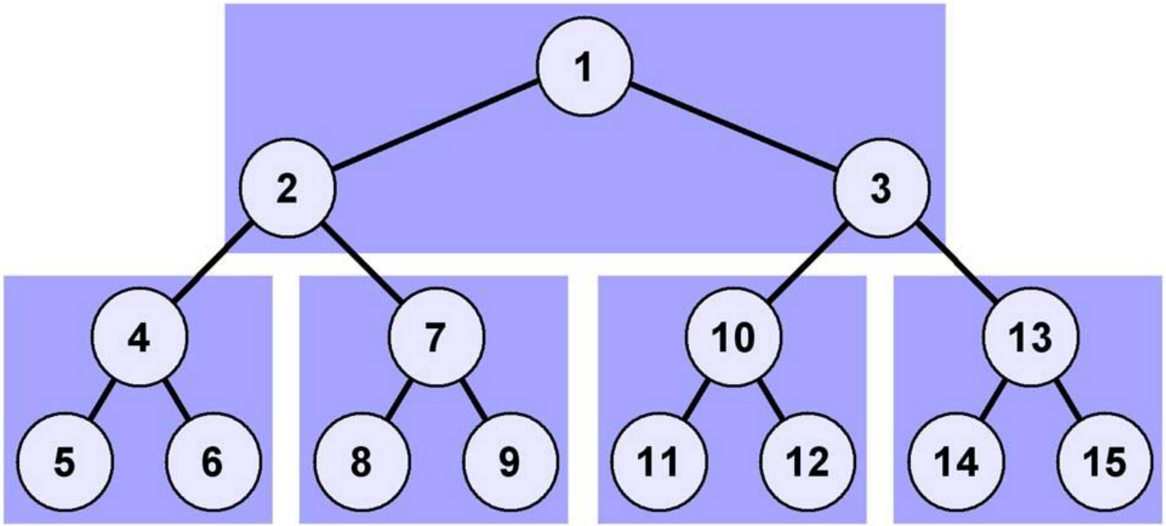
Binary Tree Example - Memory Layout

- 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
- Discussion: which is the most optimal layout to check if the given list of values is present in the tree?
- Example in the command line



Binary Tree Example - Memory Layout

- Van Emde Boas layout



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 miss

Binary Tree Example - Memory Layout

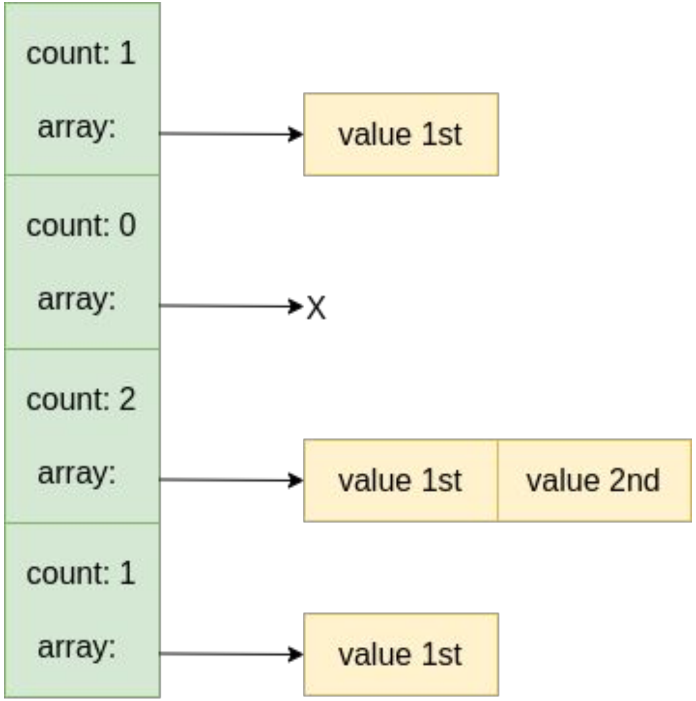
- We gain performance if:
  - We take advantage of the prefetcher
    - If two nodes are adjacent in the tree are also adjacent in memory, we increase the likelihood that the prefetcher gets activated
    - If this is the case, we get the access with no cache miss
  - We keep the *struct node* as compact 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 better cache line use

Binary Tree Modification

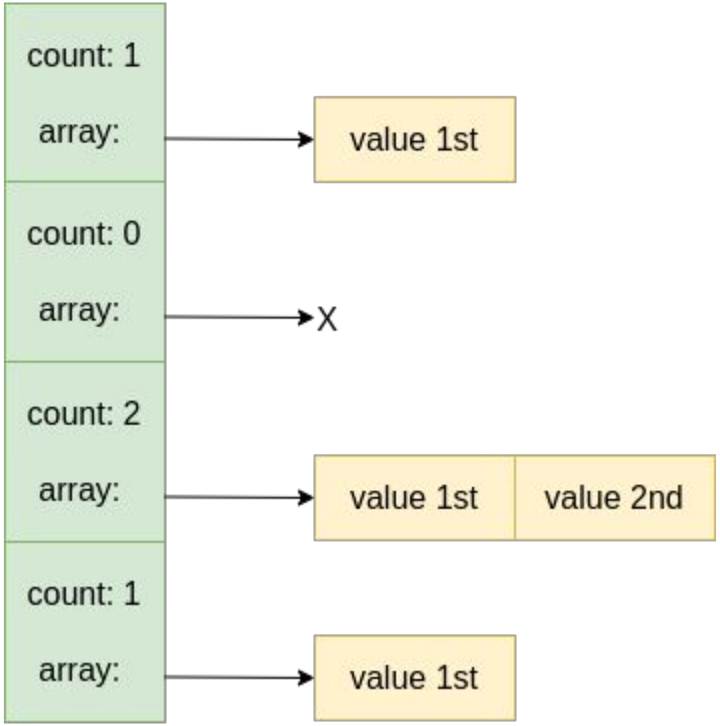
- Adding and removing nodes slowly makes the memory layout less and less optimal
- After some time, the access becomes slower and slower
- Solutions:
  - Recreate the data structure which is optimal again or
  - Perform *defragmentation* of the existing data structure or
  - Don’t delete the nodes. Keep 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

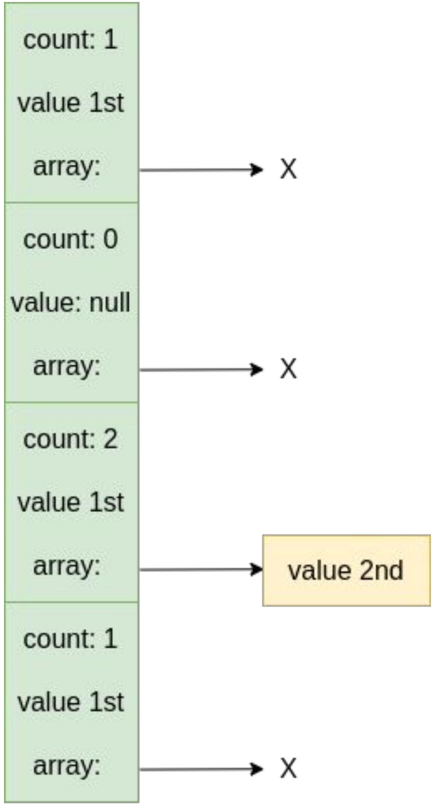
Hash Map Example

- Hash map uses an array to store it 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





https://johnysswlab.com, @johnysswlab ivica@johnysswlab.com

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

Final Words

- On modern day systems, the memory bottleneck is the thing that often limits the speed of your program
- Careful design can help mitigate some of those problem if the performance is important
  - Prefer vectors of values whenever possible
  - Object oriented design is not performance friendly
    - Possibility of many cache misses due to scattered data
    - Branch prediction misses due to polymorphism
  - Industries where performance (notably gaming) is of critical importance use different approach *data-oriented design*
- 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

Survey

Help us make this talk better. A short survey:

https://www.surveymonkey.com/r/D7FHDZ5

The End

- Most material taken from articles from 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