



# Advanced filtering

And how it came to exist

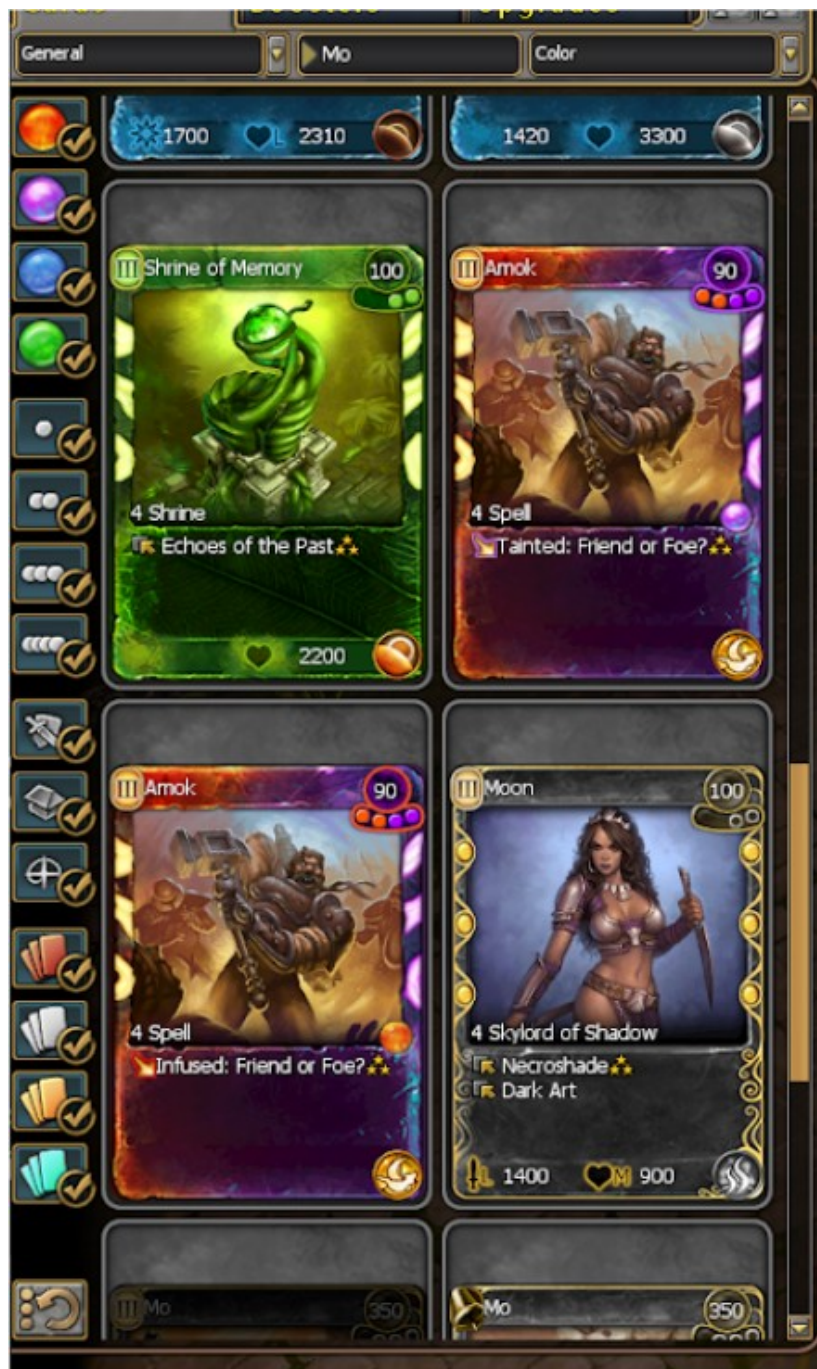
## Summary

History of advanced filtering .....	3
How it works - user's point of view .....	7
<b>Basic filtering</b> .....	7
Simple name filter .....	7
Promo .....	7
Text search for name, ability, and class name .....	9
Damage, Health, Upgrades, Charges, Copies .....	11
Size and Counter .....	13
Affinity .....	13
Work in Progress: Color .....	14
Work in Progress: Class.....	14
<b>Advanced filtering</b> .....	15
Combining filters together .....	15
<b>Auction house</b> .....	16
<b>Languages</b> .....	17
Inventory Sorting (technical part) .....	18
Nom parsing.....	22
Filtering Inventory .....	24
Filtering Auction House .....	25
Expected Questions.....	27

# History of advanced filtering

Hey everyone, I'm Kubik, a developer for Skylords Reborn. Over the past few months I have been working on improving the sorting options for Skylords Reborn. In this document I will explain how the current system works, and what will be improved in the future.

The sorting has bothered me ever since I first did a search for Mo during the EA times. Below you have a great example of how bad the current sorting is:



Here you can see there are 14 cards in the results before I get the card I am looking for, even though I wrote the whole name exactly without a typo. Additionally, because Mo does not have a color, it's also not possible to use a color filter to narrow down the search.

The example below shows the result I wanted; the actual card I was looking for.

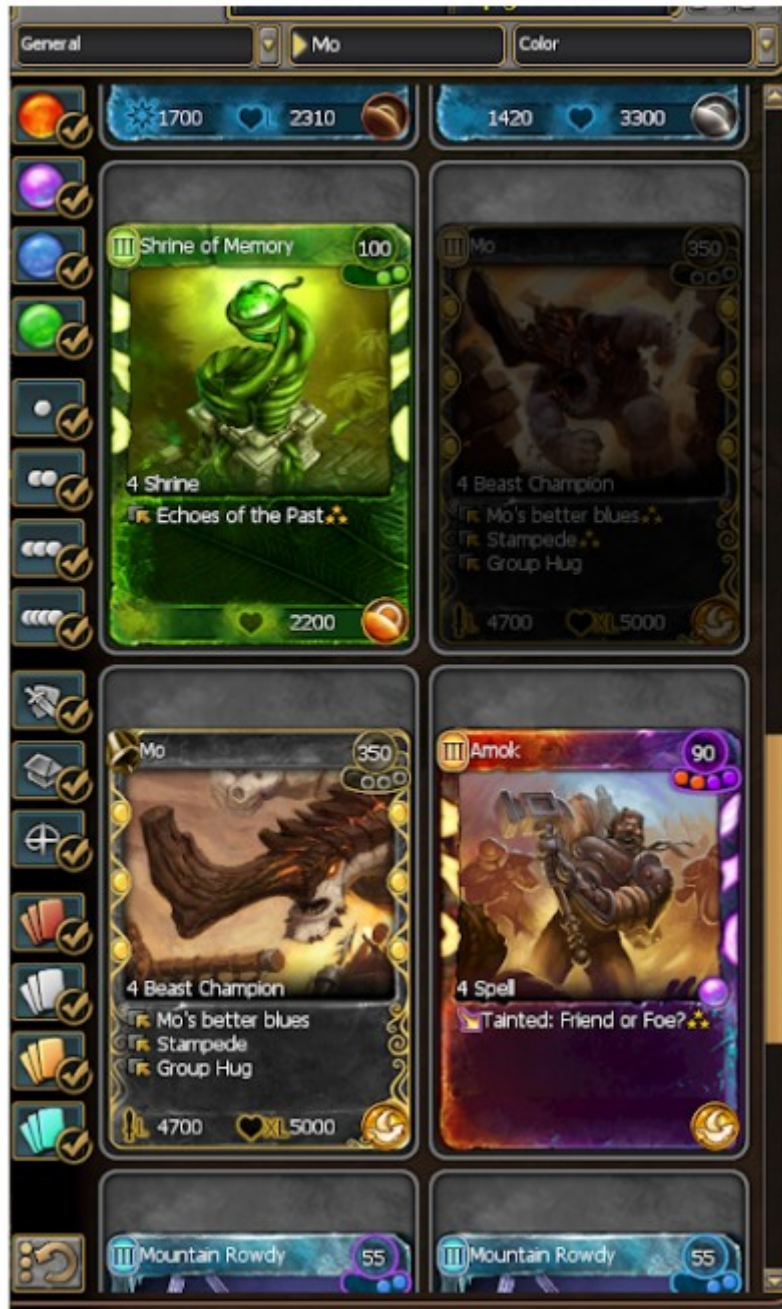


In May 2021 I finally started reversing the function that does the filtering. Which is a terrible function to reverse engineer by the way, but more on that later. I do not want to scare you off right at the start. With this I was able to create a function that behaves in more or less the same way, which meant I could start adding advanced stuff.



In June I added a function that prepares texts when Skylords Reborn is loaded, to be more specific card names and ability names in the currently selected language.

Filtering has one really wanted feature that is useful for all kinds of players: sorting. In the example of "Mo", you could see that a search put all colorless/neutral cards at the bottom, resulting in you not getting the expected result.



This is how the original search for Mo looked like.

There were numerous bug reports of “disappearing cards” when in fact the card was just not where players were looking for it. Sorting function was actually much easier to implement (and EA’s version was actually pretty fast to their standards, but was providing incorrect results).

Which brings us to the end of the historical part.

In November the first version of really advanced filtering was available for closed beta testing. It started with some basic functions: it could search for any cardname that contains the string, or is exactly the string. The same was added for abilities.

After feedback from testers the functions quickly grew and new filtering options were added. The result was that the simple way of parsing the search string became obsolete, and nom was needed to eat the string. More about that in another chapter.

One of the more interesting feature requests was having this filtering for Auction House searches as well, which is now also ready :)

By now the filtering structure has evolved that the logic no longer mimics the EA's filtering, but totally replaces it.

# How it works - user's point of view

If you type any search text that does not contain any of these symbols: “: = ? ! > <” the filtering will work as before, just faster. You will probably not notice it is faster, because this part is way below 1s. But now to the more interesting part where I show the new features.

## Basic filtering

### Simple name filter

“=Cardname”

If you prefix the text with = (equal sign) it will search for an exact match. Example: “=Mo” will not find any other card.

“?Cardname”

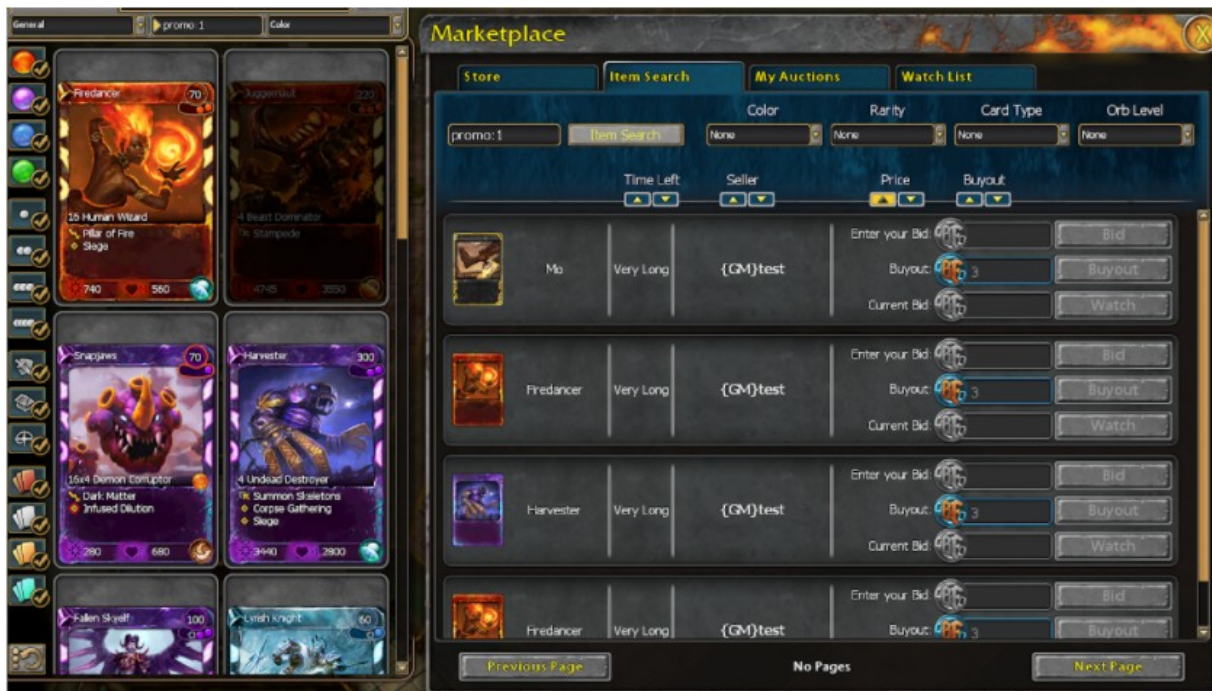
If you prefix the text with ? (question mark) it will search with error tolerance. If you know the card name almost exactly but you are not sure about spelling, you can use this.

You can also use ?1 (question mark followed by a single digit number) to specify the error tolerance you want. If you do not specify the tolerance level, level 3 will be used.

For example “?Light blade” will find “Lightblade” even though the search input had a space in it. In this example, “?1Light blade” would also work.

### Promo

“promo:1” will show you all the promos.



“promo:0” will show you all the cards that are not promos. This is useful for combinations, which we will get into in a later chapter.



## Flying, Ranged, Melee

“flying:1” and “flying:0” will do the same for flying (and non-flying) units.

The same is true for “ranged:1”, “ranged:0”, “melee:1”, “melee:0” for ranged and melee. In all these cases, 1 shows you all the cards that match, while 0 excludes them.





## Text search for name, ability, and class name

“Name:search input” is equal to just “search input”, but the prefix is needed when combining multiple filters. More about that in a later chapter.

“name:search input”, “name=search input”, “name?search input”, etc. works the same as described above for the simple name filter.

**Note:** Going forward, “:text”, “=text”, and “?text” will be called by a single term:

“\_matching\_string”



This example shows “name\_matching\_string”

The examples below show “[ability\\_matching\\_string](#)”, “[class name\\_matching\\_string](#)”  
 Name and ability are probably clear. Class name is the reference on the bottom part of the card image (next to the number of uses a card has, and above its abilities).





## Damage, Health, Upgrades, Charges, Copies

Another interesting category: numbers! We will be using “[\\_number\\_range](#)” which is defined as follows:

(**N** and **X** are placeholders for numbers)

- “>**N**” values greater than **N**
- “>=**N**” values greater or equal to **N**
- “<**N**” values smaller than **N**
- “<=**N**” values smaller or equal to **N**
- “=**N**” exactly **N**
- “:**N**-**X**” or “:**N**..**X**” values greater or equal to **N**, but smaller, than **X**
- “:**N**..**X**” values greater or equal to **N**, but smaller, or equal to **X**



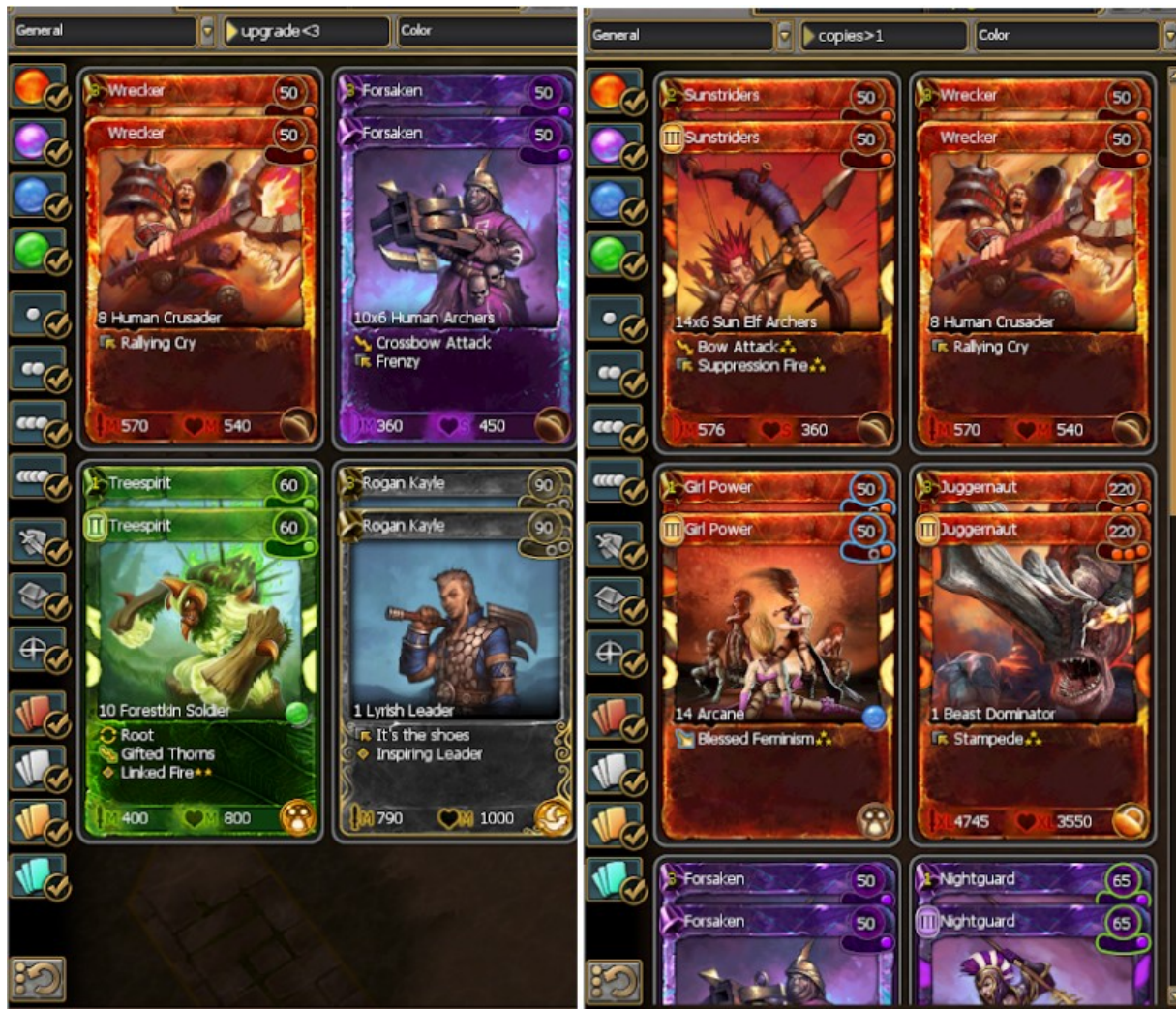
Ever wondered which cards are the most powerful? Wonder no more!

“[damage\\_number\\_range](#)” can help with just that; simply type “[damage>7000](#)”, You can find the weakest cards with this as well! :)

Want the most healthy unit? “[health\\_number\\_range](#)” can help with that, just type “[health>7000](#)”, And again; it can find the most fragile cards too :)

Looking to fully upgrade all cards? “[upgrade\\_number\\_range](#)” solves it. By using “[upgrade<3](#)” you will see all cards that are missing the upgrade.

The same applies for charges with “[charges\\_number\\_range](#)”, you can type “[charges<3](#)” to find cards that are in need of more charges.



Want to know if you have some spare cards to sell? “[copies\\_number\\_range](#)” comes to the rescue! Just type “[copies>1](#)” and what remains will be cards with a spare copy. Note: charges that are already applied are not considered a spare copy.



## Size and Counter

Do you want a big monster or a small army? “size:s”, “size:m”, “size:l”, and “size:xl” will greatly help with your search.



There is also “size:?” or “size:special” that will keep buildings and for some reason “Thunder Wagon”. Yes it really has its own armor class in game files.

Are you lacking a counter to some size and do not want to alt+tab to find out what your options are? Great news; you no longer need to! You can type “counter:s”, “counter:m”, “counter:l”, “counter:xl”, and “counter:?” or “counter:special” to find out which units can do the job.

## Affinity

Use “affinity:shadow”, “affinity:nature”, “affinity:frost”, “affinity:fire” to find cards with a specified affinity.

Use “affinity:-”, or “affinity:none” to find cards without affinity.

## Work in Progress: Color

`"colors(???)"` will filter for color, but it is recommended to use buttons on the left (they are easier, and can do **most** of the things this filter can) .

`"colors(nature)"` will show all nature, and dual colors fire+nature, shadow+nature, and frost+nature.

`"colors(nature,shadow)"` will show all nature, all shadow, and dual colors fire+nature, shadow+nature, frost+nature, fire+shadow, and frost+shadow.

`"colors(neutral)"` will show all neutral (color less) cards.

`"colors(nature,!shadow)"` will show all nature, and dual colors fire+nature, and frost+nature.

`"colors(!shadow)"` will show all cards that do not require shadow orb.

`"colors(nature,!other)"` will show all nature, and nothing else.

`"colors=(nature)"` will show all nature, and nothing else.

`"colors=(fire,nature)"` will show all fire+nature, and nothing else.

Syntax is not fully explained, only a few examples are provided, from which `"colors(neutral)"` is probably the only one that makes sense to use for the average player, because it is not yet fully decided.

## Work in Progress: Class

`"class:???"` is even more complicated. The game has 56 classes, like "female", "fire", "ranged", "hero", "god", "orc", etc. The issue is they are referred to only by number, so these names are the result of a collaborative effort to figure out what each number means.

This means that for now there are only a few hardcoded (English only) options for each class. For example; `"class:57"`, `"class:female"`, `"class:Female"` (case sensitive) all mean the same.

For these reasons they are marked as Work In Progress and a full syntax is not provided, because it can change in the future.

# Advanced filtering

## Combining filters together

There is a limit of only 255 characters for the really small text field, so I do not expect too crazy combinations will be used without a GUI-update (which is planned). There are however some more options to filter with.

For example:

“(whatever other filter you have in mind)” using braces for organizing filters has no real effect on the filter itself, but can be used to group things together and have at least some visual grouping.

“!(filter)” using an exclamation mark negates the effect of the filter, so “!(promo:1)” is the same as “promo:0”. In this simple case it makes sense to use the last option, but not every filter has a negative option. Another example: “!(ability:bow)” excludes all units that have the bow ability (and crossbow because of *contains* search)

When you combine filters, you can clarify how strict you want the results to be:

“filter 1&filter 2” only cards that satisfy both filters will be included.

“filter 1|filter 2” cards that satisfy at least one of the filters will be included.

Below are three examples:

“damage>6000 & health>6000”,

“damage>7000 | health>7000”,

“promo:1 & health<1000 & health>600 & damage<1000”



You can see strong and healthy on the left image, strong or healthy on the middle, and on the right the promo's that have.. questionable stats.



## Auction house

Almost all of these filters can be used in the AH too, with a few exceptions:  
“charges\_number\_range” does not work because the AH system does not see the inventory, so it can't check which cards you have.

“Copies\_number\_range” does not make sense here, since all cards in the AH are uncharged. While it would make sense to want to search for missing copies, because of the same reason as charges, it is not possible to implement this right now.





## Languages

All these filters support all four languages (actually five if you count UK and US English as different 🗣️). Currently, everything was translated using [deepl.com](https://www.deepl.com) and was not fully checked by translators yet. Exceptions are the two WIP “`class:???`” that is not fully translated yet.

We expect all translations to be polished before releasing. If that does not happen, it might be that some translations lack context, meaning some filters will be inconsistent for the first public iteration.



That covers the features and basics of advanced filtering. However, if you are interested in the **how**, read on.

# Inventory Sorting (technical part)

If you are here you want to see the scary code, but we will start light :D  
Let's see what the decompiled function from EA looks like.

```
void __thiscall CPGMCardPool::sort(CPGMCardPool *this, eastl_card_list
*cards_to_be_sorted)
{
    eastl_node *end; // esi
    eastl_node *begin; // ebx
    void *v4; // eax
    eastl_card_list v5; // [esp-10h] [ebp-40h] BYREF
    eastl_card_list *v6; // [esp+10h] [ebp-20h]
    eastl_list Src; // [esp+14h] [ebp-1Ch] BYREF
    char v8[4]; // [esp+20h] [ebp-10h] BYREF
    int v9; // [esp+2Ch] [ebp-4h]

    eastl_list::sub(&Src, this->base.sort_mode);
    v6 = &v5;
    v9 = 0;
    eastl::alloc((eastl_list *)&v5, Src.end - Src.begin, (int)v8);
    end = Src.end;
    begin = Src.begin;
    v4 = memcpy(v5.begin, Src.begin, (char *)Src.end - (char *)Src.begin);
    eastl::sort_list(
        cards_to_be_sorted->begin,
        cards_to_be_sorted->end,
        v5.begin,
        (int)v4 + 4 * (end - begin),
        (int)v5.after_allocated_end);
    if ( Src.begin )
        FreeMemory(Src.begin);
}
```

As you can see, decompilation went really well. But what on earth is it doing, you might ask?

Well, after I figured out their functions, I changed them to readable names, instead of just function addresses (for example, "*CPGMCardpool::sort*" instead of "*sub\_123456*") and it became a bit clearer. They are doing the following:

- The first function call knows what sorting mode you have selected. However, it will always sort by all properties, but the order varies. Meaning it allocates an array and fills it with all properties it will be sorting by, in reverse order. For example, if you have selected color then color is last in that array.
- Second and third function for whatever reason copies the list of sorting criteria ( 🤖 only EA and God know why, and I am actually not even sure if EA knows)
- And the last one does an actual sort.
  - (That one is the real mess, but I said I will not scare you too much just yet)

- It, for some reason, is so generic it starts to pass down functions as arguments, and by the point you get to actual logic, it no longer knows what it is sorting.
- It has a function to get an unknown object (by allocating a new copy of it), it has a function to get a comparison key from that object (again creating a copy, and this function changes based on which step of comparison it is doing, last one being the one you selected), it has a function to do comparison, and a function to do the swap.

Well my function to do this is not so generic, and can sort only cards, but it does so fast. When I was still not sure about the inputs, I implemented [Bubble sort](#). went with the most simple version without any early termination, which is the slowest sort I can write, but unfortunately not the slowest I ever seen.

Bubble sort has a unique advantage over all the faster sorts: it is so simple that it can easily be debugged, even in assembly, to figure out bugs like pointer to pointer to thing versus pointer to thing.

This was one of the few moments I was surprised by the game's performance in a positive way, because the original sort from EA, even though it was not always returning correct results, actually outperformed the Bubble sort by a few percent. That means BattleForge must be using something better than Bubble sort, because it sorts the cards multiple times.

I am a lazy person, so I will not be sorting cards multiple times, just to see the result of each intermediate sort get totally scrambled by the next iteration. Instead I will sort by the wanted sort mode first, and only if the cards are the same by that criteria move on to the next criteria.

Once I confirmed and understood the inputs, and could sort it correctly, I switched to library sort [slice::sort\\_by](#). At this point, performance was no longer in EA's favor; the new sorting method is, in most cases, more than 1000 times faster.

As you can see on the next page, my function is longer BUT includes all the logic in less than 100 lines.

Here is how my function works:

- line 9 converts the list to Rusts primitive type called [slice](#)
- line 10 obtains reference to global game's resource manager
- lines 12-28 converts requested sort mode to const array reference of sorting priorities (no allocation needed)
- line 29 a call library function [slice::sort\\_by](#) that will use fast algorithm to sort and might get even better without any effort on our side (like if 39µ wouldn't be fast enough, on my 5 years old PC 🤖)
- lines 30-91 provide function that that compares 2 cards
  - lines 30 and 31 dereference the pointers
  - line 32 is for loop, but it has an early termination on line 85. In case the first comparison criteria is already different, it will not be looping at all
  - lines 89-91 are there just in case all of the above comparison criterias would be the same, affinity should not be
  - lines 58-62 compare by name, by resolving name resources for the card and comparing them
  - lines 53-57 compare by power requirements
  - lines 32-82 are mostly collapsed to fit on the page, but they follow a very similar pattern as the name, and power

That is about all that there is to say about sorting, we leverage the power of the Rusts standard library, and do only simple type conversions to match the types.



```

5 pub fn sort(
6     this: &'static CP6MCardSelection,
7     cards_to_sort: *mut CardList
8 ) {
9     let cards_to_sort : &mut [*mut Card] = unsafe {&mut *cards_to_sort}.as_slice_mut();
10    let main : &mut Main = Main::get_instance();
11
12    let cmp_by : &[SortMode] = match this.sort_mode_8 {
13        SortMode::Orb => &[SortMode::Orb, SortMode::Color, SortMode::Class, SortMode::Power,
14            SortMode::Rarity, SortMode::Edition, SortMode::Alphabetical],
15        SortMode::Color => &[SortMode::Color, SortMode::Orb, SortMode::Class, SortMode::Power,
16            SortMode::Rarity, SortMode::Edition, SortMode::Alphabetical],
17        SortMode::Class => &[SortMode::Class, SortMode::Color, SortMode::Orb, SortMode::Power,
18            SortMode::Rarity, SortMode::Edition, SortMode::Alphabetical],
19        SortMode::Power => &[SortMode::Power, SortMode::Color, SortMode::Orb, SortMode::Class,
20            SortMode::Rarity, SortMode::Edition, SortMode::Alphabetical],
21        SortMode::Alphabetical => &[SortMode::Alphabetical, SortMode::Color, SortMode::Orb,
22            SortMode::Class, SortMode::Power, SortMode::Rarity, SortMode::Edition],
23        SortMode::Rarity => &[SortMode::Rarity, SortMode::Color, SortMode::Orb, SortMode::Class,
24            SortMode::Power, SortMode::Edition, SortMode::Alphabetical],
25        SortMode::Id => &[SortMode::Id],
26        SortMode::Edition => &[SortMode::Edition, SortMode::Color, SortMode::Orb, SortMode::Class,
27            SortMode::Power, SortMode::Rarity, SortMode::Alphabetical],
28    };
29    cards_to_sort.sort_by(|c1 : &*mut Card, c2 : &*mut Card| {
30        let c1 : &Card = unsafe {&**c1};
31        let c2 : &Card = unsafe {&**c2};
32        for cmp_by : &SortMode in cmp_by {
33            let r : Ordering = match cmp_by {
34                SortMode::Orb => {...}
35                SortMode::Color => {...}
36                SortMode::Class => {...}
37                SortMode::Power => {
38                    let c1 : &Card = main.card_by_id( id: c1.id_with_upgrade());
39                    let c2 : &Card = main.card_by_id( id: c2.id_with_upgrade());
40                    c1.power_cost().cmp( other: &c2.power_cost())
41                }
42                SortMode::Alphabetical => {
43                    let c1 :String = main.card_by_id( id: c1.id_with_upgrade()).name();
44                    let c2 :String = main.card_by_id( id: c2.id_with_upgrade()).name();
45                    c1.cmp( other: &c2)
46                }
47                SortMode::Rarity => {...}
48                SortMode::Id => {...}
49                SortMode::Edition => {...}
50            };
51            if r != std::cmp::Ordering::Equal {
52                return r;
53            }
54        }
55    });
56
57    let c1 : &Card = main.card_by_id( id: c1.id_with_upgrade());
58    let c2 : &Card = main.card_by_id( id: c2.id_with_upgrade());
59    c1.affinity_power_name_c0.cmp( other: &c2.affinity_power_name_c0)
60 }

```

# Nom parsing

---



[Nom](#) is a great library that allows easy parsing of anything.

Please check out [“parsing code link”](#) before the explanation below:

- lines 1-6 is the top level function “parse\_expr” that accepts text as an input and return “Result” (either success with the parsed expression, or parsing errors)
  - This function does two things: either parse the whole expression, or just the simplified version of name search
- line 9 “AttackArmor” enum type for size/counter filtering
- line 11 “ShouldInclude” enum type, just because I like enum more than a bool, because the name is more expressive
- lines 13-17 “FilterConfig” enum type for string comparisons
- lines 19-41 “Expr” enum type that holds the final filtering expression, names here more or less match the names in [How it works - User's point of view](#) so no need to explain them here again

This whole parsing is very inspired by nom's test/example [nom's test arithmetic ast](#), because this expression with parentheses, and/or operators is kind of similar to mathematical expression. It just has much more composable blocks that replace mathematical “Value”.

Because of this, I will go over only the more interesting functions, since most of them just repeat with different names.

- lines 70-73 “string” is an interesting function, because card name can contain spaces, and based on language also weird characters, so I needed a way to define string that does not consume special characters that would be part of another composed expression
- lines 75-82 is the most interesting one; it basically says that any of the listed things can be inside of composed expression
- lines 108-121 “affinity” says that if the text starts with “affinity:” or any of the other three language alternatives and is followed by an affinity name, it creates an expression that matches that affinity. It uses a number instead of a nicer enum, because of compatibility with EA’s other “logic”.
- lines 123-174 “colours” and “color” which allow “colors” in both US, and UK English and other languages. However, the inner colors are only allowed to be in English, so it is still a Work In Progress. On line 140 it asks for the opening brace and on line 160 it checks if the provided color was last and is followed by a closing brace.
- lines 176-191 “cfg\_str\_matching” is the function responsible for exact matching string or fuzzy matching it
- lines 193-212 just have prefix and call “cfg\_str\_matching”
- lines 214-219 is the WIP “class”
- lines 258-298 “numbers\_range” parses number range as mentioned earlier in [Damage, Health, Upgrades, Charges, Copies](#)

And that's about it for parsing the search string! There is definitely some space for optimization in that code, but then we are talking about milliseconds for the filtering, even for really long filters, so I'm not in a hurry to improve that. :)

# Filtering Inventory

[Warning: Decompiled code with mess :\)](#)

If you ignored the warning and opened the link above, you know that this one did not decompile so great as sorting (well in sortings case it was because the actual logic was not there, just top level function)

If you can see the filter hierarchy in this, you should really consider checking out our [open staff positions](#), because we really lack people for this. Even if you are just barely able to find it, please check the [open staff positions](#) help is always welcomed! :)

```
pub fn filter(this: &'static mut CGUICollectionView, filtered_cards: *mut CardList,
all_cards: &'static CardList)
{
    let c_res_main = Main::get_instance();
    let filtered_cards = unsafe { &mut *filtered_cards};

    let combined_filter = filters_to_card_filter(&this.filters_map.all());
    trace!("{:?}", combined_filter);

    filtered_cards.zero();
    for card in unsafe{all_cards.iter().map(|c| &**c)}{
        if !card.is_mastercard.val() { continue }
        let c_res_card = c_res_main.card_by_id(card.id_with_upgrade());
        if apply_card_filter(Some(card), c_res_card, Some(all_cards), &combined_filter){
            filtered_cards.insert(card as *const _ as *mut _);
        }
    }
}
```

Let's start with a comparison; here it is much more easily visible what this function is doing. My function starts by converting pointer types to Rust-friendly references, then it converts EA's filter hierarchy (filters on the left + search string) to a single expression. Here is a default one with empty text:

```
And([
    Or([Type(Squad), Type(Building), Type(Spell)]),
    Or([
        Color { include: 48, exclude: 0 },
        Color { include: 34, exclude: 0 },
        Color { include: 40, exclude: 0 },
        Color { include: 36, exclude: 0 }
    ]),
    Or([TokenCount(1), TokenCount(2), TokenCount(3), TokenCount(4)]),
    Or([Rarity(1), Rarity(2), Rarity(3), Rarity(4)])
])
```

After that, the function iterates over all cards, ignoring all that are not the mastercard (additional copies). Only looking at the mastercard results is performance improvement for everyone with a large collection of duplicates. After that it just applies the filter and if the card should stay, it will copy the pointer to the output list.



# Filtering Auction House

Some would guess this will be a similar function, but unfortunately the function for AH works a bit differently: parse the totally different filtering structure, with caching of the result, then send a list of cards that filter matched to the server, wait for response, parse the response, and update the UI.

I will not even bother sharing a terribly decompiled function of that size (it has terrible allocation issues) because [IDA](#) has for some reason (with this function specifically) big issues allocating overlapping variables on the stack. They overlap, because they are used in different parts of the functions. The result is around 1000 lines of incorrectly named, and typed variables, and when any is manually fixed, all others break :(

As you can see on the next page, my version is not that much different. It converts pointers to Rust-friendly types. Building the filter is a bit more complicated, because empty text does not mean to include all.

However, after that it just goes through all 5000+ cards in the game and smartly ignores non playable cards (which EA's version does not, meaning searching for neutral + UR + 1 orb will ask the server for two cards and the server needs to ignore them). Well, not anymore!

After that it asks the server. It looks like it also asks for boosters, but we drop that field on network request, meaning at least it is empty, which was not with EA's version. If the server responds with an error, just pass that error up. Otherwise convert the returned auctions to GUI auctions, add them to the list, and done.



```

fn ah_search(this: *mut GetAuctionManager, search_text:*mut CString, page_size:u32,
page:u32, sort_by:u32, descending:u32) -> u32 {
    unsafe{
        let this = &mut*this;
        let search_text = &*search_text;
        let main = Main::get_instance();
        let (cards, _backing_field) = if
            this.color_filter == 0 && this.orb_level_filter == 0
            && this.type_filter == 0 && this.rarity_filter == 0 && search_text.size == 0 {
            (List::new(), None)
        } else {
            let filter = build_filter(this, search_text);
            trace!("{:?}", filter);
            let cards = main.cards.iter().filter_map(|c|
                if CardTemplate::from((&c.val).base_id()) != CardTemplate::NotACard
                && apply_card_filter(None, &c.val, None, &filter) {
                Some((&c.val).base_id() as u64)
            } else { None }
            ).collect::<Vec<u64>>();
            if cards.is_empty() {
                (List::new(), None)
            } else if cards.len() >= 500 {
                warn!("can not search for more than 500 cards, but trying to look for: {}
with filter: {:?}", cards.len(), filter);
                (List::new(), None)
            } else {
                (List::from_ptr(cards.as_ptr(), cards.len()), Some(cards))
            }
        };
        let boosters = List::new();
        let rmr = &*ask_server(this.shop_gateway &cards, &boosters, sort_by,
            descending, page, page_size, 0, 0, 0);
        if rmr.result != 0 {
            return rmr.result;
        }

        this.total_auction_count = rmr.total_found_auctions;
        init_auctions(this.auctions_begin, this.auctions_end);
        this.auctions_end = this.auctions_begin;
        let auctions = std::slice::from_raw_parts(rmr.auctions as *const CNetAuctionVO,
rmr.auction_count as usize);
        for a in auctions {
            let card_id = *(a.cards_ptr as *const u32);
            let auction = AhCard {
                id: a.id,
                start_bid: a.start_bid,
                buyout: a.buyout,
                current_bid: a.current_bid,
                remaining_time: a.remaining_time,
                seller: a.seller.data.clone(),
                card_id: card_id,
                card_name:
main.card_by_id(card_id).base.strings.find(2).unwrap().to_c_string(),
                highest_bidder_id: a.highest_bidder_id
            };
            auctions_add(&mut this.auctions_begin, &auction);
        }
        drop_rmr(rmr)
    }
    0
}

```

# Expected Questions

---

## **When will this feature go Live?**

This feature went live alongside the Anniversary Patch on the 18th of December 2021.

## **Will more filters be added?**

Yes, we will continue to add more filters in the future.

## **I spotted a bug/possible improvement/missing feature, where to report?**

You can message me directly on discord **Kubik#9154**

## **I like the code, and I would like to help, what to do?**

[Follow the White Rabbit](#) 

## **Any other questions/comments?**

Please use discord, for things like this communication is much easier than on the forums.