APC-Library

Hi all, this is our 2nd Advanced topic for Advanced Programming Concepts. For this topic we are going to dive head-first into three types of standard library types, the `pair`, `map` and `vector` type.

We have all used various types of containers provided by the C++ standard library before. These containers are the foundation of data storage and manipulation in our C++ programs. The C++ standard library takes away the need to set these containers up yourself.

Where this is good on one hand, the fact that these container types are given to us ready to be used can cause you to make mistakes in implementing the containers, as you might not fully understand the basics behind them.

To help you to better understand these container types, we will dive into `std::pair`, `std::vector` and `std::map` of the C++ standard library. We will be looking at the GNU ISO C++ Library specifically. We will first look into how these containers are implemented in the GNU ISO library, after which we will implement our own (basic) version of the containers.

The GNU ISO C++ Library

`std::pair`

`std::pair` is a special, simpler type of `std::tuple`. Where std::tuple doesn't have a fixed number of member variables, `std::pair` only ever has two.

The GNU ISO C++ library states the basis of `std::pair` as follows:

```
  template<typename _T1, typename _T2>
    struct pair
    : private __pair_base<_T1, _T2>
    {
        typedef _T1 first_type;    ///< The type of the `first` member
        typedef _T2 second_type;   ///< The type of the `second` member

        _T1 first;                 ///< The first member
        _T2 second;                ///< The second member

    public:
        //Many member functions
    };
```

Here, `first` and `second` are the pair's private member variables of types `_T1` and `_T2`. This part of the pair class is still fairly straight forward.

The template shows a struct that holds two variables of types `_T1` and `_T2`. `std::pair` has several basic member functions, such as a number of constructors among which:

```
#if __cplusplus >= 201103L
        constexpr pair(const pair&) = default;  ///< Copy constructor
        constexpr pair(pair&&) = default;       ///< Move constructor

        // DR 811.
        template<typename _U1, typename
        enable_if<_PCCP::template
        _MoveCopyPair<true, _U1, _T2>(),
                bool>::type=true>
        constexpr pair(_U1&& __x, const _T2& __y)
                : first(std::forward<_U1>(__x)), second(__y) { }
```

These are a move and copy constructor which take a different pair and construct a pair copying or moving the `first` and `second` of the passed pair. There is also a template for a constructor that takes two values and sets up the pair with these two values. Where the move and copy constructor are pretty readable, the third constructor has a lot more generic template-shenanigans going on. These are necessary to make a universal library that works with all systems, but it may cause a user who wants to take a quick look into the library to understand what is going on to be overwhelmed, and preventing them from properly understanding what is actually going on.

`std::pair` doesn't need any get functions, as the member variables `first` and `second` are public.

Lastly `std::pair` has several swap functions, one of which looks as follows:

```
 /// Swap the first members and then the second members.
_GLIBCXX20_CONSTEXPR void
swap(pair& __p)
noexcept(__and_<__is_nothrow_swappable<_T1>,
        __is_nothrow_swappable<_T2>>::value)
{
    using std::swap;
    swap(first, __p.first);
    swap(second, __p.second);
}
```

As we can see in this example, `std::pair::swap()` makes use of `std::swap()` to swap the member variables of one pair with those of a second pair. There is a `noexcept()` with some generic statements that may, again, be hard for a novice to understand.

We will guide you through implementing your own simplified version of `std::pair` later on. But first we will look into `std::vector` and `std::map`.

`std::vector`

Secondly, we will be looking at `std::vector`. One could say `std::vector` is just a glorified array, which is true to some extent. `std::vector` makes use of a base struct `_Vector_base` which looks like this:

```
  template<typename _Tp, typename _Alloc>
    struct _Vector_base
    {
      typedef typename __gnu_cxx::__alloc_traits<_Alloc>::template
    rebind<_Tp>::other _Tp_alloc_type;
      typedef typename __gnu_cxx::__alloc_traits<_Tp_alloc_type>::pointer
        pointer;

      struct _Vector_impl_data
      {
      pointer _M_start;
      pointer _M_finish;
      pointer _M_end_of_storage;

      //some functions (bases for copy & swap)
      };

      //some functions (bases for std::vector functions)
    }
```

In this base struct, `_M_start`, `_M_finish` and `_M_end_of_storage` are defined. These are pointers of type `__gnu_cxx::__alloc_traits<_Tp_alloc_type>::pointer`, which can be seen as simple pointers to the first and last element in the array, and to the end of the storage, which can be seen as `_M_start + capacity`. The use of this base struct is quite confusing if you are trying to quickly look into the header to gain some understanding of the logic behind `std::vector`.

Because the data is stored in this `_Vector_base` struct, the class `vector` serves as a wrapper expanding on the functionality of the `_Vector_base` struct. the `vector` class adds functions like a bunch of constructors, `begin()` and `end()` functions, as well as some data-altering functions such as `insert()`, `erase()`, `push_back()` and `pop_back()`.

All of these functions refer back to the `_Vector_base` struct, as can be seen in the `capacity()` function below:

```
  size_type
  capacity() const _GLIBCXX_NOEXCEPT
  { return size_type(this->_M_impl._M_end_of_storage
         - this->_M_impl._M_start); }
```

In our own implementation of the `vector` header, we will get rid of this `_Vector_base` subclass and focus on simplicity to help you understand how `vector` works at its base.

`std::map`

At its base, `std::map` looks like this:

```
  template <typename _Key, typename _Tp, typename _Compare = std::less<_Key>,
       typename _Alloc = std::allocator<std::pair<const _Key, _Tp> > >
    class map
    {
    public:
      typedef _Key                  key_type;
      typedef _Tp                   mapped_type;
      typedef std::pair<const _Key, _Tp>        value_type;
      typedef _Compare                  key_compare;
      typedef _Alloc                    allocator_type;

    public:
      class value_compare
      : public std::binary_function<value_type, value_type, bool>
      {
        {...}
      };

    private:
      /// This turns a red-black tree into a [multi]map.
      typedef typename __gnu_cxx::__alloc_traits<_Alloc>::template
    rebind<value_type>::other _Pair_alloc_type;

      typedef _Rb_tree<key_type, value_type, _Select1st<value_type>,
               key_compare, _Pair_alloc_type> _Rep_type;

      /// The actual tree structure.
      _Rep_type _M_t;

    public:
      //A lot of constructors and functions
```

We can see that `_M_t` is the main member variable of type `_Rep_type`, which is a Red-Black tree made up of `std::pair` nodes. A Red-Black tree is used in `std::map` because it is a fast type of container that is easy to maintain and to keep sorted.

A place where this is very clearly visible, is in the `begin()` function of `std::map`:

```
//stl_map.h:
    iterator
    begin() _GLIBCXX_NOEXCEPT
    { return _M_t.begin(); }

//stl_tree.h:
    iterator
    begin() _GLIBCXX_NOEXCEPT
    { return iterator(this->_M_impl._M_header._M_left); }
```

Here we can see the first `begin()` function of `std::map` that has to return an iterator to the first element. Because the data is stored in a Red-Black tree, the left-most element of this tree has to be returned. This is what the second `begin()` function is for. It returns an iterator to the left-most element of the tree.

In our simplified version of the `std::map` we will use a vector instead of a Red-Black tree. This way we can make use of our own `mc::vector` and `mc::pair` classes to create the `mc::map`.

The MC library

To further illustrate, we will now implement the functionalities of `std::pair`, `std::vector` and `std::map` in simple C++ classes ourselves. This library will be the MC library, named after its creators (Marnix & Camiel).

`mc::pair`

Starting off with `pair`, we will create a header file `pair.h` and create a basis for the `pair` class in the `mc` namespace:

```
namespace mc {

    template <typename T1, typename T2>
    class pair {

    public:
        T1 first;
        T2 second;
    };

}
```

We declare a template with typenames `T1` and `T2` for the ability to handle any variable/container type. We declare two member variables `first` and `second` of type `T1` and `T2` respectively. These variables are public just like in the `std::pair` implementation.

Now that we have a place for data to be stored, let's make some constructors that can actually get data in our pair.

Constructors

Let's start the constructors off with the default and parameterized constructors:

```
// Default constructor
pair() = default;

// Parameterized constructor
pair(T1 first, T2 second) :
    first {first},
    second {second}
    {
        //Nothing to do here
    }
```

and a copy constructor:

```
// Copy constructor
pair(const pair& other):
    first{other.first},
    second{other.second}
    {
        //Nothing to do here
    }
```

Similar to constructors, we might want to do something like:

```
auto d = mc::make_pair(1, 2.4);
```

For this we will implement a `make_pair` function as follows:

```
template <typename T1, typename T2>
pair<T1, T2> make_pair(T1 first, T2 second) {
    return pair{first, second};
}
```

With these constructors we can execute commands like the following:

```
mc::pair<int, int> a{};     // sets first and second to 0
a.first = 1;              // a == (1, 0)
a.second = 2;             // a == (1, 2)

mc::pair<int, std::string> b{1, "hello"};   // b == (1, "hello")
mc::pair<int, std::string> c{b};            // c == (1, "hello")

auto d = mc::make_pair(1, 2.4);
```

We can manually print these values to the console to check if the constructors worked correctly

```
std::cout << "a: (" << a.first << ", " << a.second << ")\n";
std::cout << "b: (" << b.first << ", " << b.second << ")\n";
std::cout << "c: (" << c.first << ", " << c.second << ")\n";
std::cout << "d: (" << d.first << ", " << d.second << ")\n";
```

This generates the following output:

```
a: (1, 2)
b: (1, hello)
c: (1, hello)
```

That works splendidly!

Swapping

Let's now add a swap function, so we can swap the values of two pairs:

```
void swap(){
    using std::swap; // enable 'std::swap' to be found
    swap(first, second);
    return *this;
}
```

One thing to keep in mind is that this function only works if the types T1 and T2 of both pairs match. This is the same with `std::pair`, so we will keep it at this, as we are not here to improve on the library, we are just here to increase our understanding of it.

Another thing that might stand out while looking at the swap function is that we are using std::swap. The reason for this is further explained in this stack overflow question.

Comparators

We may very well like to compare the pairs at some point. We shall now implement some operators that will be able to handle the comparison of two pairs (of the same type).

`operator<` and `operator>` can be member functions, defined as follows:

```
// < compare operator
bool operator<(const pair<T1, T2>& other) const {
    if (first == other.first) {
        return second < other.second;
    }
    return first < other.first;
}

// > compare operator
bool operator>(const pair<T1, T2>& other) const {
    if (first == other.first) {
        return second > other.second;
    }
    return first > other.first;
}
```

These functions will compare `first` of both pairs first. In case these are the same, `second` will decide the result of the comparison.

Example:

```
mc::pair<int, int> a{1,2};
mc::pair<int, int> b{1,3};
mc::pair<int, int> c{2,1};

std::cout << (a<b) << ", " << (a>c) << ", " << (b>c) << std::endl;
```

```
1, 0, 0
```

Looking at the output we can see that `a` is indeed smaller that `b` (1 == 1, 2 < 3), `a` is not greater than `c` (1 < 2) and `second` is correctly ignored in the comparison.

Lastly `b` is not greater than `c` (1<2) and again `second` is correctly ignored.

Another comparator, maybe an even more important one, is the equal operator, or `operator==`. This one looks as follows:

```
template <typename T1, typename T2>
bool operator==(const pair<T1, T2>& a, const pair<T1, T2>& b) {
    return (a.first() == b.first() and a.second() == b.second());
}
```

An interesting thing we can see here is that the `operator==` takes two pairs. This is because it is declared **outside** of the `pair` class.

Example:

```
mc::pair<int, int> a{1,2};
mc::pair<int, int> b{1,2};
mc::pair<int, int> c{2,1};

std::cout << (a==b) << ", " << (a==c) << ", " << (b==c) << std::endl;
```

```
1, 0, 0
```

These outputs speak for themselves. We can see that a(1,2) and b(1,2) are equal, whereas c(2,1) is different.

The inequality operator is automatically generated by the compiler if operator== is defined. (Since C++20) See https://en.cppreference.com/w/cpp/language/operators for more information regarding this operator overloading

Stream operator

Lastly we will implement a function based around quality of life improvement. A `operator<<` overload for `std::ostream` will allow us to print the contents of our pair more easily. It will look like this:

To be able to use this in any scenario, such as file writing, we also want a separate `std::ostream::operator<<` overload. It will look as follows:

```
// Out stream operator for pair
template <typename T1, typename T2>
std::ostream& operator<<(std::ostream& stream, pair<T1, T2>& other) {
    stream << "(" << other.first << ", " << other.second << ")";
    return stream;
}
```

The function above will again be declared **outside** of the `pair` class. It simply takes an output stream and a pair and calls the pair's `print()` function, of which we already know the workings.

Example:

```
mc::pair<int, int> a{1,2};
mc::pair<int, int> b{1,3};
mc::pair<int, int> c{2,5};

std::cout << "a" << a << ", b" << b << ", c" << c << std::endl;
```

```
a(1, 2), b(1, 3), c(2, 5)
```

Of course there are some more functions to the `std::pair` class that we won't go into in this report. We have gained a sufficient understanding of how a `pair` works and what we can do with it. Time to move on to `mc::vector`!

`mc::vector`

We will start off our vector class the same way we started our pair class:

```
namespace mc {

    template <typename T>
    class vector {

    private:
        pointer m_data;
        std::size_t m_cap;
        std::size_t m_sz;
    };
}
```

Here m_data is our raw array holding the data. As we can see m_data is of type `pointer`, this is defined in the class (among with other definitions to probe the vector class easily):

```
using value_type = T;
using pointer = T*;
using const_pointer = const T*;
using reference = T&;
using const_reference = const T&;
```

This means that m_data is a pointer to the first element of type T in the array. The vector class keeps track of the amount of elements used in the array and will automatically grow the array when the capacity is almost used. This is explained below

Constructors

We will now set up the constructors, which will look as follows:

```
// Normal constructor
explicit vector(std::size_t capacity):
    m_data{ new T[capacity] }, // thanks to @zaldawid
    m_cap{ capacity },
    m_sz{ 0 } {}

static constexpr std::size_t DEFAULT_CAP{20};

// Default constructor
vector(): vector(DEFAULT_CAP) {};

// Initializer list constructor
vector(std::initializer_list<T> list) : vector(DEFAULT_CAP) {
    for (auto& entry : list) {
        push_back(entry); // This will update size while pushing back entries
    }
}

// Copy constructor
vector(const vector& other):
    m_data{ new T[other.capacity()] },
    m_cap{ other.capacity() },
    m_sz{ other.size() } {

    // copy over data from other vector
    std::uninitialized_copy(other.begin(), other.end(), m_data);
}
```

We tried using the `static_cast<value_type( ::operator new( capacity * sizeof(value_type) ) )` memory assignment for our pointer. The advantage of this is that `::operator new()` just allocates raw memory, nothing else. We prefer this method of allocating memory because no object construction should take place at the constructor of our vector. There is also a static cast because `::operator new()` returns a `void*` (generic pointer). The compiler will be unable to bind this generic pointer to our value_type pointer.

We did not use this memory assignment method because we were running into issues, and we did not have the time left to debug those issues since we have exams. This is definitely interesting, and we will hopefully come back to this in the future.

With these constructors we can set up vectors in the main like:

```
mc::vector<std::string> a{};            // a() empty with capacity 20
mc::vector<int> b{10};                  // b() empty with capacity 10
mc::vector<int_wrapper> c{1,2,3,4,5};   // c(1, 2, 3, 4, 5)
mc::vector<int_wrapper> d{c};           // d(1, 2, 3, 4, 5)
```

We can manually print the values in the vector to the console to check if the constructors worked correctly:

```
for (std::size_t i = 0; i < c.size(); i++)
    std::cout << "c[" << i << "]: " << c[i] << "\n";
```

This generates the following output:

```
c[0]: 1
c[1]: 2
c[2]: 3
c[3]: 4
c[4]: 5
```

That works well! But this can get very repetitive when printing vectors often. We have a solution for this! You can read about that solution below.

Accessing data

What is a data structure worth if you can't access any of the data? Not a dime we think. So let's implement some functions that will let us access the data of our `mc::vector`.

Starting off, we will set up a `begin()` and `end()` function.

```
const_pointer begin() const noexcept {
    return m_data;
}

const_pointer end() const noexcept {
    return &m_data[m_sz];
}
```

These functions are very simple. They simply return a pointer to the first and last element in the array respectively.

We shall also implement an `operator[]` function to handle access by index. It will look like this:

```
reference operator[](std::size_t index){
    return m_data[index];
}
```

This function is not much more complex. Since the raw array that is used in our vector already has functionality for using index accessing, we can simply use that and return the m_data on the given index.

Inserting data

We obviously also want to be able to easily add data to the vector. Three functions we will implement for this are the `insert`, `push_back` and `pop_back` functions. These functions will look as follows:

```
void push_back(reference& entry) {
    adjust_cap();
    m_data[m_sz++] = std::move(entry);
}

void insert(const std::size_t index, const value_type entry) {
    if (m_sz == 0 or index >= m_sz) {
        push_back(entry);
        return;
    }

    adjust_cap();

    for (size_t i = m_sz; i >= index; i--) {
        m_data[i + 1] = m_data[i];
    }

    m_data[index] = entry;
    m_sz++;
}
```

`push_back` is very straight forward. It updates the capacity of the vector and adds the entry to the back of the array. `insert` is only slightly more complex. It first checks whether the index is in range(if it's not, it will use `push_back`) after which it has to shift the elements of the array to the right and overwrite the data at the given index with `entry`.

Adjusting the capacity efficiently takes its own function. It is described below.

Keeping track of capacity

```
static constexpr std::size_t GROWTH_FACTOR{2};
void adjust_cap(std::size_t how_many_extra_elements = 1) {

    std::size_t required_capacity = m_sz + how_many_extra_elements;

    if (required_capacity > m_cap) {
        std::size_t new_capacity = m_cap;

        // Calculate new capacity
        while (new_capacity <= required_capacity)
            new_capacity *= GROWTH_FACTOR;

        pointer replacement = new T[new_capacity];

        // Move over contents of array to replacement
        std::uninitialized_move(begin(), end(), replacement);

        // Destroy left over elements
        std::destroy_n(m_data, m_sz);

        // Delete old memory
        delete[] m_data;

        m_data = replacement;
        m_cap = new_capacity;
    }
}
```

This functions checks if the amount of elements we want to store can fit in our current capacity. If our array is not big enough to store new elements, we calculate a new capacity and grow the array. We grow the array by defining a new array, copying over the contents in our current array to the replacement array. After the copy, we destroy the old objects (this calls the destructor of the elements in our array) and we free up the memory. We then set our m_data variable to point to this new piece of memory we just prepared and update the capacity.

Erasing data

Finally, in the trend of data manipulation, we have erasing data. We will implement a `pop_back` and an `erase` function to allow erasure of data from the vector. These functions will be implemented as follows:

```
void pop_back() noexcept {
    std::destroy_at(m_data + m_sz - 1); // std::destroy_at calls the destructor of the object pointed to by p, as if by p->~T()
    --m_sz;
}

void erase(std::size_t index) {
    if (index >= m_sz) {
        // Index is out of bounds
        throw "vector_error: erase(i) is out of bounds\n";
    }

    for (std::size_t i = index; i < m_sz - 1; i++){
        m_data[i] = m_data[i+1];
    }

    m_sz--;
}
```

`pop_back` is fairly straight forward. It removes the last element in the array and decreases the size by one. `erase` is only slightly more complex. it first checks whether the passed index is in range(if it isn't, it throws an exception) after which it shifts the data in the array left, getting rid of the data to be erased in the process.

Stream operator

Lastly we will implement a function based around quality of life improvement. A `operator<<` overload for `std::ostream` will allow us to print the contents of our vector more easily. It will look like this:

To be able to use this in any scenario, such as file writing, we also want a separate `std::ostream::operator<<` overload. It will look as follows:

```
// Out stream operator for vector
template <typename T>
std::ostream& operator<<(std::ostream& stream, vector<T>& other) {
    stream << "{";

    for (std::size_t i = 0; i < other.size(); i++) {
        stream << other[i];

        // Add ', ' between every element except the last
        if (i != other.size() - 1)
            stream << ", ";
    }

    stream << "}";

    return stream;
}
```

The function above will again be declared **outside** of the `vector` class. It simply takes an output stream and a vector and prints out the content of the vector in a way that you would write a vector declaration with an initializer list into a C++ program.

Example:

```
mc::vector<int_wrapper> c{1, 2, 3, 4, 5, 6, 7, 8, 9};
mc::vector<int_wrapper> d{c}; // d{1, 2, 3, 4, 5, 6, 7, 8, 9}

std::cout << d << "\n";
```

Gives the following output:

```
vector{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Using `mc::vector` with std algorithms

The standard has some great algorithmic functions, like `std::sort`! These take in an iterator to your data type and take away the need for implementing your own sorting, accumulation or other function. These iterators are commonly retrieved by calling `.begin()` and `.end()` or `std::begin()` and `std::end()` on an object.

Keeping track of capacity

```
 static constexpr std::size_t GROWTH_FACTOR{2};
void adjust_cap(std::size_t how_many_extra_elements = 1) {

    std::size_t required_capacity = m_sz + how_many_extra_elements;

    if (required_capacity > m_cap) {
        std::size_t new_capacity = m_cap;

        // Calculate new capacity
        while (new_capacity <= required_capacity)
            new_capacity *= GROWTH_FACTOR;

        pointer replacement = new T[new_capacity];

        // Move over contents of array to replacement
        std::uninitialized_move(begin(), end(), replacement);

        // Destroy left over elements
        std::destroy_n(m_data, m_sz);

        // Delete old memory
        delete[] m_data;

        m_data = replacement;
        m_cap = new_capacity;
    }
}
```

This functions checks if the amount of elements we want to store can fit in our current capacity. If our array is not big enough to store new elements, we calculate a new capacity and grow the array. We grow the array by defining a new array, copying over the contents in our current array to the replacement array. After the copy, we destroy the old objects (this calls the destructor of the elements in our array) and we free up the memory. We then set our m_data variable to point to this new piece of memory we just prepared and update the capacity.

`mc::map`

This class stores a list of key value pairs. We will start off our map class the same way we started our vector and pair classes:

```
namespace mc {

    template<typename TKey, typename TValue>
    class map { // This whole class is a wrapper around an mc::vector<mc::pair>
    public:

        using first_type = TKey;
        using first_pointer = TKey*;
        using first_reference = TKey&;
        using first_const_reference = const TKey&;

        using second_type = TValue;
        using second_pointer = TValue*;
        using second_reference = TValue&;
        using second_const_reference = const TValue&;

        using pair_template = mc::pair<TKey, TValue>;
        using pair_template_pointer = pair_template*;
        using pair_template_reference = pair_template&;
        using pair_template_const_reference = const pair_template&;

        using vector_template = mc::vector<pair_template>;
        using vector_template_pointer = vector_template*;
        using vector_template_reference = vector_template&;
        using vector_template_const_reference = const vector_template&;

    private:
        vector_template m_vector;
    };
}
```

Here m_vector is our underlying data structure. As we can see m_vector is of type `vector_template`, this is defined in the class (among with other definitions to probe the vector class easily) to be:

```
using vector_template = mc::vector<pair_template>;
```

Where `pair_template` is the following:

```
using pair_template = mc::pair<TKey, TValue>;
```

So, in full, this is a variable of type `mc::vector<mc::pair<Tkey, TValue>>`

Constructors

We will now set up the constructors, which will look as follows:

```
    // Default constructor
    explicit map() : m_vector {} {}

    // initializer list constructor
    map(std::initializer_list<pair_template> list) : map() {
        for (auto& entry : list) {
            push_back(entry);
        }
    };

    // Copy constructor, this will call vector's copy constructor which will handle the rest

    map(const map &other) : m_vector {other.raw()} {}
```

With these constructors we can set up maps in the main like:

```
// Initializer list constructor
mc::map<char, std::string> test{
    {'a', "apple"},
    {'g', "giraffe"},
    {'w', "wonderland"}
};

test.push_back({'c', "cat"});

// Copy constructor
mc::map copy = test;
```

To print the values in the map, we can call the following:

```
std::cout << copy << "\n";
```

This generates the following output:

```
mc::vector{(a, apple), (g, giraffe), (w, wonderland), (c, cat)}
```

It prints `mc::vector` because map is really just a wrapper around vector.

Accessing data

To access the data in our map, we can implement a `begin()` and `end()` function as well as an `operator[]`, similarly to in our `mc::vector`. In all fairness, we can simply refer to the `mc::vector` implementation of these functions as the data in our map, at its base, is stored in an `mc::vector`. With this in mind, the functions will look as follows:

```
pair_template* begin() {
    return m_vector.begin();
}

pair_template* end() {
    return m_vector.end();
}

[[maybe_unused]] pair_template_reference operator[](std::size_t index) {
return m_vector[index];
}
```

We have already seen how and why these functions work in the part on `mc::vector`, so we don't need to explain them again here.

Inserting data

The same as with accessing the data in the map counts for inserting data. The implementations of `mc::vector` are called by the functions in `mc::map` These look as follows:

```
// Push back function for manual mc::pair<T1, T2>
void push_back(const pair_template entry) {
    m_vector.push_back(entry);
}

// Push back function for pushing back values of template types
void push_back(const first_type first, const second_type second) {
    m_vector.push_back(pair_template(first, second));
}

// Insert function for manual mc::pair<T1, T2>
void insert(const std::size_t index, const pair_template entry) {
    m_vector.insert(index, entry);
}

// Insert function for inserting values of template types
void insert(const std::size_t index, const first_type first, const second_type second) {
    m_vector.insert(index, pair_template(first, second));
}
```

Note that we have two different functions for `push_back` and for `insert`. This allows for multiple ways of calling these functions such as:

```
mc::map<char, std::string> test{};

mc::pair<char, std::string> doggo{'d', "dog"};

test.push_back(doggo);          //This uses the top of the two push_back functions stated in the code block above
test.push_back({'c', "cat"});   //This uses the top of the two push_back functions stated in the code block above
test.push_back('a', "ape");     //This uses the bottom of the two push_back functions stated in the code block above
```

Keeping track of capacity

Capacity management gets taken care of by the vector when we push back! The map class does not have to do this.

Erasing data

For erasing data from the map, just like with inserting data, we refer to the functions of `mc::vector`. The functions will then look as follows:

```
void pop_back() {
    m_vector.pop_back();
}

void erase() {
    m_vector.erase();
}
```

These functions speak for themselves. If you forgot how these functions worked, you can check the part about `mc::vector` of this report.

Stream operator

Lastly we will implement a function based around quality of life improvement. A `operator<<` overload for `std::ostream` will allow us to print the contents of our vector more easily. It will look like this:

To be able to use this in any scenario, such as file writing, we also want a separate `std::ostream::operator<<` overload. It will look as follows:

```
// Out stream operator for map
template<typename TKey, typename TValue>
std::ostream& operator<<(std::ostream& stream, map<TKey, TValue>& other) {
    stream << other.raw();
    return stream;
}
```

The function above will again be declared **outside** of the `map` class. It simply takes an output stream and a map and writes the underlying vector to the stream. The vector will handle the outputting to this stream. This will allow us to print the entire map at once.

Complete `mc::map` example:

```
// Initializer list constructor
mc::map<char, std::string> test{
    {'a', "apple"},
    {'g', "giraffe"},
    {'w', "wonderland"}
};

mc::pair<char, std::string> doggo{'d', "dog"};
    test.push_back(doggo);
    test.push_back({'c', "cat"});

test.insert(3, 'b', "banana");

// Copy constructor
mc::map copy = test;
copy.sort();

std::cout << copy << std::endl;
```

```
mc::vector{(a, apple), (b, banana), (c, cat), (d, dog), (g, giraffe), (w, wonderland)}
```

That works splendidly. And with all three containers set up and working, that's it!