implement vector from scratch - my notes

One important thing I should say is the original std::vector uses allocator to deal with memory management, and to me this seems like some kind of smart pointer, or something that abstracts the heavy lower details of memory management, making it easier to work with. I opted for `::operator new` because it would force me to work with the memory at a lower level (not as low as assembly though), and therefore would be a very good exercise in refreshing my memory on how to work with memory management in C++.

Overall I think it's okay code but I don't like it. It could be improved and shortened. It started off pretty clean and easy to implement, but grew as I had to handle more cases.

- inconsistency with variable naming
- bad variable naming like private fields why size_ instead of just size. But that's because the function name is size in the original std::vector, so I had to change the variable name in the private fields to avoid name conflicts (the compiler complained about this)
- I'm pretty sure some things can be done in fewer lines of code, for instance the resize function feels unnecessarily convoluted
- While I have done alot of happy path testing and trivial sad paths, I still haven't tested every possible scenario that can break the vector implementation. This feels very tedious. For instance I found one example that was non-trivial that could lead to memory leak if the type is non-primitive in my resize method, so that's something I have to fix. It only occurs when implementing the size * 2 behavior for resize when expanding, there may be cases where the space between size *and capacity* is default initialized, so when size_ expands into these regions the code will reinitialize objects and overwrite the old ones without destroying them, leading to memory leak
- extra mental load in the memory management functions, it's not immediately clear why some things are done the way they are, only becomes clearer once you identify the common scenarios which you need to handle
- I think some logic can be refactored into helper functions. I tried this with some, but didn't find that many that I could do it with
- resize was probably the worst to implement, part of it due to misunderstanding how it worked and that there were some non-trivial edge cases to handle
- I feel like alot of the functions can instead of accepting copies of arguments (invoking their copy constructor) make use of move semantics OR accept references. This would make it a bit more efficient and cleaner

Notes about implementation

- **Edge cases**
  alot edge cases not taken care of, like if user gives overflow of number or negative numbers etc
- **Memory allocation logic**
  memory allocation is a bit janky, when new memory is needed I just delete old memory and then allocate to an arbitrary available memory address. Perhaps this is the only way as there is no way to know in advance if the immediate memories are not already occupied.
- **Avoid implicit conversion as opposed to std::vector**
  I didn't want the implicit conversions when accepting numbers for size of vector or indexing, so I made sure to check that they are positive integral types. In std::vector that is not the case, they just do implicit conversion to an integral type.
- **Allocate memory according to capacity**
  It seems that std::vector always allocates memory according to the capacity size, which is double the size of the actual elements. If dealing with primitive types, these extra allocated memories (beyond the original elements) will be initialized to default values for the primitive types, like 0 for int for instance. If dealing with custom classes they will initialize these allocated memories to whatever the default constructor does for these classes, for std::string it will be empty string.
- **Design choice capacity**
  There's a design choice here for capacity at initialization. One could in advance assume that user will add more elements, in which case it would be prudent to initialize the capacity to double the size of the elements. I did this, but it seems that std::vector initializes capacity to exactly the size. I found out that it would be more pain if I did it the way I opted for, because it would give uninitialized objects at these extra allocated memories, so when I wanted to implement push_back, it would not be clear if I could just overwrite these memory addresses, or if I needed to initialize them first and then fill with values. That's why I switched to the way std::vector sets the capacity at initialization, whether it be a copy constructor or a move constructor or some other constructor.
- **Resize in std::vector?**
  Resize to a size, if the size is bigger than capacity of vector then it will expand memory to that size AND initialize the additional memory locations with default values of the class that it contains. If it already have some elements, these are also added or transfered idk exactly, but they are added to the new memory locations in addition to the default values. The old values comes first and then the default values. This means if you now call `push_back(1000)` the element will be

pushed to the last position in the memory, which would be at index `new_capacity - 1`. Example to make this easy to understand:

```
  std::vector<int> t{1,2,3};
  t.resize(10);
  t.push_back(1000);  // this will be found at index (10 - 1) = 9, NOT
at index (4-1) = 3
```

So we can see that resize doesn't just allocate memory, it actually initializes objects on those memory locations as well, filling up the vector with default values.

After resize following scenario can happen
size and new cap will always be set to the same value regardless. The question is what value will the capacity be set to?

1. $\text{new cap} < \text{capacity}$ and $\text{new cap} < \text{size}$ => size set to new cap and capacity remains the same. This will cause shrinking, so destructor needs to be invoked on the appropriate objects.
2. $\text{new cap} < \text{capacity}$ and $\text{new cap} > \text{size}$ => size set to new cap and capacity remains the same. Default values initialized on the appropriate memory locations if they are empty.
3. $\text{new cap} == \text{capacity}$ and $\text{new cap} > \text{size}$ => size set to new cap and capacity remains the same. Default values initialized on the appropriate memory locations if they are empty.
4. $\text{new cap} == \text{capacity}$ and $\text{new cap} == \text{size}$ => nothing happens
5. $\text{new cap} > \text{capacity}$ => size and capacity will become the same and set to the value of new cap. Reallocation of memory occurs.
6. if $\text{new cap} > \text{capacity}$ and $\text{size} * 2 < \text{new cap}$ =>size and capacity will both be set to new_cap value.
7. if $\text{new cap} > \text{capacity}$ and $\text{size} * 2 \geq \text{new cap}$ =>size set to new cap and capacity will be $\text{size} * 2$.

I will probably skip implementing this behavior of size * 2, because resize is already ugly and will become uglier if I do. I don't think it matters that much for the toy examples I use the vector for so whatever.

```
  std::vector<int> t{1,2,3,4,5,6};  // capacity will be 6
  t.resize(10);  // expansion required, will take 6 * 2 = 12, so new
capacity will be 12. Size will be 10, because default initialization
until the desired size 10.
  t.resize(3);  // shrinking, capacity stays the same i.e. 12, but
destroy objects until desired size is reached, size will be 3.
```

```
  t.resize(11);  // no expansion because capacity is 12, but 11 is
bigger than current size 3, so need to default initilize until size 11
is reached. If primitive then only need to default initialize from 10
to 11 (index 9 to 10), because primitive don't have destructors, so
they were never destroyed, but instead remained in the memory
```

- **Push_back vector**
  Push_back doesn't default initialize values when the size is equal to the capacity meaning more space is needed. It will reallocate memory, but not initialize the objects on the memory.

- **Primitive types destructors**
  There are some cases when you try to remove something, but the type is primitive. Then they can't be destroyed by their destructor because they don't have one. Usually what happens is that they remain in memory, but the size of the vector is decremented or something, such that the user can't access them. I've seen this by experimenting with std::vector.

- **Reallocate or not reallocate memory**
  Something to think about in general for the methods implemented. Usually if you still have allocated memory, but not initialized, and the elements to be added fits, then don't reallocate new memory somewhere else, use the already allocated ones. If not, then will need to reallocate to somewhere else decided by the OS.
- **Why not use new T[] to allocate memory?**
  `new T[]` allocates + initializes the objects. Also once you use this kind of new you can't use `::operator delete`, so you can't mix the usages of `new` or `new T[]` and `::operator new`. Because I don't think there's a way to tell which one you've used, it's better to be consistent here. And since std::vector has methods that don't always initialize objects on memory directly after allocating memory (some methods only allocates memory), it's better to go for `::operator new` for flexibility. Obviously this leads to more code, but less headache.

- **Why sometimes use iterator to get the pointer of the class? Why not just use .data()?**
  I could do that since assigning to pointers is just shallow copy, but there are some methods that require iterator so that's why I added the iterator method. And I initially assumed the private ptr field to not be accessible through any methods, so that's why I saw Iterator as the only way to acquire it. But then later on in the documentations I saw that for std::vector there is a data() that returns the ptr of the class, but at that point it was too late. So now I can use either .data() or .begin() to acquire the pointer of the vector class. There's also a third method, just access the pointer in the private because member functions always have access

to private members in a class, and then do shallow copy. I think this way would have been the easiest and most straightforward.

- **Why not use T& more as argument type in functions to avoid copying unecessarily?**
  I had some error when doing that so I stopped, I think it was because I tried to do with `const T&`. Also, while some functions in the original std::vector requires & as argument type I found it a bit janky to let the user have to provide a reference of a variable in order to use the function, so I skipped for those functions as well. But it's true that I should use it more to avoid invoking copying of objects so much, can be expensive if they are big objects.
- **Copy assignment in vector does a resize if the other vector is bigger**
  I tested this with some experiments, size and capacity becomes exactly the same if bigger, and accessing outside this array will lead to undefined behavior or random values.
  If shrinking then size will be smaller but capacity remain the same.

- **When vector is initialized capacity and size will be the same**
  At initialization capacity and size is always the same. It's only later on when adding or resizing or something else that changes the memory so that if it's not enough it will expand and increase the capacity. The common way to do it is by doubling the current capacity.

- **Move assignment for std::vector**
  Tested with som experiments. The impression I got is that it just switches the pointer to point to the new vector.

```
        std::vector<int> s104{1,2,3,4,5,6};
        std::vector<int> s105{1,2,3,4,5,6,7,8,9};
        s104 = std::move(s105);
        assert(s104.size() == 9);
        assert(s104.capacity() == 9);
        assert(s104[8] == 9);
        assert(s104[5] == 6);
        std::cout << "s104[9]: " << s104[9] << std::endl;

        std::vector<int> s106{1,2,3,4,5,6};
        std::vector<int> s107{1,2,3,4,5,6,7,8,9};
        s107 = std::move(s106);
        for (int e : s107) {
                        std::cout << e << " ";  // prints out the
content of s106
        }
```

```
        assert(s107.size() == 6);
        assert(s107.capacity() == 6);
        assert(s107[5] == 6);
        assert(s107[4] == 5);
        std::cout << "s107[9]: " << s107[9] << std::endl;  // gives
random values
        std::cout << "s107[8]: " << s107[8] << std::endl;  // random
values
```

- **My custom copy method**
  My copy method is weird and only for initialization. It accepts only iterators, because that's what I thought they did, but apparently they accept pointers. Since std::iterator can be used as a pointer it seems to be able to convert it, but my custom iterator class currently cannot. So instead i accept iterators. But it would have been clearer if it accepted pointers I think. The T* type approach would just have the body as below, it's very simple.

```
for (int i=0; i < other.size_; ++i) {
    elements[i] = other.elements[i];
}
```

TODOS

- should probably add more try and catch for when i actually allocate memory and something happens, then need to destroy the allocated memory before throwing an error?
- Should use std::move more instead of copying
- resize, there is one edge case, when the new cap is bigger than size but not bigger than capacity. If we have this case, wheere default values have been initialized, so we don't want to reinitialize when we resize:

```
  std::vector<int> t{1,2,3,4,5,6};  // capacity will be 6
  t.resize(10);  // expansion required, will take 6 * 2 = 12, so new
capacity will be 12. Size will be 10, because default initialization
until the desired size 10.
  t.resize(3);  // shrinking, capacity stays the same i.e. 12, but
destroy objects until desired size is reached, size will be 3.
  t.resize(11);  // no expansion because capacity is 12, but 11 is
bigger than current size 3, so need to default initilize until size 11
is reached. If primitive then only need to default initialize from 10
to 11 (index 9 to 10), because primitive don't have destructors, so
they were never destroyed, but instead remained in the memory
```

The crux of the problem is sometimes when capacity is bigger than size, default values will be stored in the memory locations between size and capacity, but when you expand size to these memory location, if not cautious you will do double initialization which is a problem if type is non-primitive, because they will be overwritten but not destroyed by the destructor (I think).

```
  std::vector<int> t{1,2,3,4,5,6};  // capacity will be 6
  t.resize(10);  // expansion required, will take 6 * 2 = 12, so new
capacity will be 12. Size will be 10, because default initialization
until the desired size 10.
  t.resize(3);  // shrinking, capacity stays the same i.e. 12, but
destroy objects until desired size is reached, size will be 3.
  t.resize(11);  // no expansion because capacity is 12, but 11 is
bigger than current size 3, so need to default initilize until size 11
is reached. If primitive then only need to default initialize from 10
to 11 (index 9 to 10), because primitive don't have destructors, so
they were never destroyed, but instead remained in the memory
```

Here I reinitialize with `new` regardless if the object already exists in the memory location. This would mean that I overwrite already existing objects until I reach 11 from size 3, but the old objects are not destroyed properly because I currently don't call the destructors, which would lead to memory leak I have to assume. This wasn't a trivial case, because it depends on what you do when you default initialize to meet the size requirement, if you just naively overwrite objects the old ones might not be properly destroyed.

Same situation but with different elements and sizes can be illustrated like this

```
Initialize: std::vector<int> t(8)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
              |
         size, capacity
Default initialized from 1 to 8.

Expand: t.resize(11)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
                     |              |
                   size          capacity
Default initialized from 1 to 16.

Shrink: t.resize(5)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
        |                           |
      size                       capacity
```

```
Destroyed objects stored between 6 and 11.

Expand: t.resize(13)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
                           |        |
                         size    capacity
Reinitialize objects stored in 6 to 13. But 12 and 13 were already
initialized (when we did t.resize(11)), so we are overwriting them, but
not properly destroying them (invoking their destructors), which causes
memory leak.
```

But once again this is fine, because I didn't implement the size * 2 behavior, so it will never happen.

- Implement insert and delete (at arbitrary positions)
- test more in general or use it more, that way can gradually improve.
- Write proper test cases with catch2.

Random things I learnt

Allocate memory of custom class type - new vs ::operator new

new

`::operator new` (the global new operator) only allocates raw memory, it doesn't initialize at all. This made it so I couldn't assign an existing object to them for some reason, as they needed to be initialized through a loop.
`new` operator instead allocates memory + initializes the objects by calling their default constructor. So now you can start overwriting them if you want in situations like copy constructor or something else.

delete

In practise if you use `::operator new` then you can only use `::operator delete` to delete the object, but it won't delete the entire array or memory allocation, you have to loop through it manually and delete the objects initialized there by for insance invoking their destructors for each one of them. If you however used `new` then you can just call `delete` like usual and it will take care of invoking the destructors. If you allocated array of memory with `new[]` then need to use `delete[]` to properly delete.

Class templates in header files

Apparently class templates should normally be initialized in header files.
If you however want to separate declaration from definition, then need to include the

file at the end of header file like `#include "something.cpp"`. There is `.tpp` extension, but vscode seems bad at recognizing it, so really annoying as can't really tell if I write incorrectly or not.

Rules of three, rules of five, rules of zero

Rules of three
If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.

Rules of five
Because the presence of a user-defined (include = default or = delete declared) destructor, copy-constructor, or copy-assignment operator prevents implicit definition of the move constructor and the move assignment operator, any class for which move semantics are desirable, has to declare all five special member functions.

Rules of zero
Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership (which follows from the Single Responsibility Principle). Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators1(https://en.cppreference.com/w/cpp/language/rule_of_three#cite_note-1). This rule also appears in the C++ Core Guidelines as C.20: If you can avoid defining default operations, do

Function name and local variable name same

In this case it will overwrite the function name, so that you won't be able to call it. However, if the function name and the local variable name was not within the same function it would have been fine. But avoid giving same name regardless because of confusion.

```
   class MyClass {
         public:
             void func() {
                 int func = 42;  // This shadows the method name 'func'
                 // func();      // This would now refer to the local
variable, not the method
             }
};
```

Avoid same namespace as method name

This is obvious but I had a namespace named main in a header file, and then in a `.cpp` file had the regular `int main()`, which then complained when I tried to compile the files.

Const order on pointer matters, but for reference doesn't matter

`const int* ptr` is a pointer to a constant integer, where the integer can't change (constant) but the pointer can change.
`int* const ptr` is a constant pointer to an integer, where the integer can change but not the pointer.
However for references it doesn't matter, because you can't change a reference (it needs to be initialized at the start).

Never use malloc if you can in c++, prefer new

Malloc is C and new is c++. Even though you can use malloc for compatibility working with C-like code, only use malloc if you have to. Malloc + free, and new + delete. Don't mix the two, as in using malloc but then using delete or new and free, because it leads to undefined behavior. New is type-safe, it will invoke the constructor for the type called with new.

Copy doesn't have boundary checks

`std::copy(source.begin(), source.end(), dest.begin())` doesn't even throw anything even if the dest container doesn't have enough space. So it will lead to undefined behavior, but you won't know what exactly as copy won't give any feedback. Best is to either make sure that the list is enough or use `std::back_inserter`.

Difference between pointer and reference?

Reference = alias, not an adress. So you can directly access the value of the variable it refers to
Pointer = points to the address of the variable. You can't directly access the value unless you dereference the pointer

- A reference can't be reassigned, so once you bind the reference (and you have to do that at initialization time) it will be referring to the variable forever. That means if you try to change the reference, all you are really changing is the value of the variable that the reference refers to.
- Reference cannot be assigned to null, have to always refer to an existing object or valid memory location

- Reference has automatic dereferencing compared to pointer that needs explicit (*) ptr
- Reference doesn't have their own memory address, compiler does some extra work to make sure that reference is just an alias or another name for the same variable it refers to. So basically, the memory of the reference is the memory of the variable it refers to.
- Only in class may reference (if it's a field for instance) need to be stored separately as part of the class memory layout. May take up memory as a pointer in that case.

But there are some interesting cases with references

- doing `&` as an operator on a variable or object will give back a **pointer** to the address of where the variable/object is.
- when return type or argument type is `&` then it's a reference, but can be treated as the original variable that had the value/object. It's basically an alias. example

```
int& func() {
    static int x = 10;
    return x;  // Returns a reference to x
}

int main() {
    int& ref = func();  // ref is now a reference to x
    ref = 20;           // This modifies the original x
}
```

Iterator classes in c++

Iterator classes are usually defined within a container class, not separately, so you usually see things like `vector.begin()`, `vector.end()`. They **behave** like pointers, in the sense that if you do the usual operations like you would on a pointer it would have the same effect on an iterator. So you can work with them as if they were pointers. Iterators are like a wrapper around a pointer, but I guess with some more functionality.

Can access pointer like arrays

Can use `ptr[i]` to access the memory address at ith step.

Why never use new T[]?

I thought about it in cases for instance when the constructor didn't accept any existing elements to copy from. But in the destructor there isn't an easy way to knowing which method was used to initialize. So it's better to be consistent I think, makes it easier.

.begin() for initializer_list<\T> always returns a pointer

For most standard containers .begin() would return an iterator (some kind of wrapper around a pointer to the underlying array that is supposed to act like a pointer). But for initializer_list<\T> it returns a pointer.

Explicit can only be used inside declarations inside the class

If separate them like .h and .cpp setup, then explicit can't be used outside to design the definition. What you do is declare that explicit inside the class and then define it outside without the explicit. It only needs to be defined at declaration level.

Cannot have conflicting member function names and members (fields)

As title says. I had a function named size and a field variable named size, didn't work.

Playing with pointers and references

```
int a = 5;
int&b = a // Ok, a is an lvalue (has a memory address)
int&t = 10 // error, 10 is an rvalue (doesn't have a memory address,
temporary value)
int* ptr = &a; // Ok, will get a pointer pointing to variable a
address, when & is used as an operator it will always return a pointer
int& t = *ptr; // Ok, dereferencing pointer to the value it points to,
but the reference variable will be able to retrieve the address of a
int& s = t; // Ok, will refer to the same variable as t, which is a

T* var = new T; // var holds the address of some dynamically allocated
object
T** ptr_to_var = &var; // ptr_to_var now holds the address of the
pointer var
```

Difference between class and typename in templates?

typename can be used when have nested types or dependent types, with class you can't.

```
// here no difference, they both are equivalent
template <class T>
void func(T arg) {
    // Do something with arg
}

template <typename T>
void func(T arg) {
    // Do something with arg
}

// but here there is a difference, typename allows nested types
(dependent types)
template <typename T> class MyClass {
        typename T::value_type myValue; // T::value_type is a type
};
```

Conventions:

- `class`: More common historically in older C++ code.
- `typename`: Often preferred in modern C++ code because it is clearer when you're dealing with types. Also, `typename` is required in some contexts (e.g., nested types), so using it consistently can help avoid confusion.

Assert

```
#include <cassert>
#include <iostream>

int main() {
    int x = 5;
    int y = 0;

    // Check that y is not zero before dividing (an assumption)
    assert(y != 0);

    std::cout << x / y << std::endl; // This line will not be reached
if assert fails

    return 0;
}
```

Assertions are only active in debug builds (when `NDEBUG` is not defined). If you compile your program with the `-DNDEBUG` flag or define `NDEBUG`, the `assert` macro is disabled, and the checks are not performed. This is useful because you may not want these checks to be performed in production builds, where efficiency is critical.

Const reference can bind to rvalues

```
int c = 10;
int& a = c;  // Ok, reference can bind to lvalue (c variable is
somewhere in the stack so it has a memory address)
int& a = 5;  // error, reference can't bind to a rvalue
const int& a = 5;  // This works because const references can bind to
rvalues
```

In this case, `a` is a reference to the temporary value `5`, but you can't modify `a` because it's a `const int&`. This is allowed because you're promising not to modify the value.

Why You Can't Assign a Non-const Reference to an rvalue
The reason for this restriction is that allowing a non-const reference to bind to an rvalue would imply that you could modify the rvalue, but modifying a temporary value like `5` doesn't make sense because it's not stored in memory with a stable address. Allowing this would create dangling references and undefined behavior.

In short:

- `int&` can bind to **modifiable lvalues** (e.g., variables like `c`).
- `const int&` can bind to **rvalues** (e.g., literals like `5`), allowing read-only access to temporary values.

Static assert

Does compile-time check and throws error if the type of the argument is not of the required time that you specify. This will only do compile-time check, not run-time check. If want to do run-time check then use if statement where you cast the argument to the desired type and check if the value are the same as the original argument after the cast.

Mutation - Functional approach or OOP

Something I thought about while implementing was

- should I return the relevant value from the function or should I just mutate it in place?
- should the function accept field as an argument or let the function directly access the field (because it's within the class)?
  In functional they usually just return and accept inputs and that is usually to promote immutability, so you don't have mutator functions. But since I was already doing OOP then I thought it would be best to take advantage of OOP paradigm. For some functions I wanted to keep them a little more general, so that they could accept arguments, in case the arguments needed to be other values than the field values.

Difference between const and constexpr

Const variables can be assigned at run-time `const int x = some_function();`. Constexpr variable must be initialized with a value that can be evaluated at compile time `constexpr int y = 5` or

```
int runtimeValue() {
    return 42;
}

constexpr int compileTimeValue() {
    return 42;
}

int main() {
    constexpr int y = compileTimeValue();  // OK: y is evaluated at
compile time
    // constexpr int z = runtimeValue();   // Error: runtimeValue() is
not constexpr
}

// another example
constexpr int add(int a, int b) {
    return a + b;  // everything inside here also has to be constexpr
}

int main() {
    constexpr int sum = add(2, 3);  // OK: computed at compile time
    int a = 5;
    // constexpr int result = add(a, 3);  // Error: a is not constexpr
}

// constexpr works with lvalues as well as rvalues
```

```
// lvalue
constexpr int x = 42;  // lvalue
constexpr const int& ref = x;  // OK: lvalue reference to a constexpr
variable

int main() {
    static_assert(ref == 42, "Compile-time check");
}

// rvalue
constexpr int getValue() {
    return 42;
}

int main() {
    constexpr int y = getValue();  // rvalue returned from the function
    static_assert(y == 42, "Compile-time check");
}
```

- `const`: Declares that a variable's value cannot be changed, but the value can be computed at **runtime**.
- `constexpr`: Ensures that the value is **evaluated at compile time** and that all function arguments and operations are constant expressions. Functions marked `constexpr` must themselves be capable of evaluation at compile time.

However one important note about const is if you accept input as const, then when calling it's functions they must also be const functions, otherwise you get error.

Neat technique with constexpr

```
// Only call destructors for non-trivial types
if constexpr (!std::is_trivially_destructible<T>::value) {
        for (size_t i = 0; i < size; ++i) {
                data[i].~T();  // Call destructor for non-primitive
types
        }
}
```

At compile time if the conditional evaluates to false, then the body within will not be included in the compiled code (it will not be generated), I guess an optimization is happening.

RAII = resource acquisition is initialization

RAII is more a guideline for initialization and destruction practises.
Constructor initializes resources on memory and destructor releases. The point is you should try to only allocate memories in the constructor (initialization stage) and free resources in the destructor.

multiple templates - special rules

You can use `template<typename T, typename S>` if you're defining a free-standing function (outside a class), or if you're adding **multiple template parameters to a class method**.

```
// class
template<typename T, typename S>
class MyClass {
public:
    MyClass(T first_value, S second_value)
        : first{first_value}, second{second_value} {}

    void display() {
        std::cout << "First: " << first << ", Second: " << second <<
std::endl;
    }

private:
    T first;
    S second;
};

// to a standalone function nothing to do with the class, but if it was
apart of it then it would have worked, because that's the template
structure for the class
template<typename T, typename S>
my_function(T t, S s) {}
```

However, for a **member function** of a class template, where you're adding an extra type `S` only for the function, you have to use the nested `template<typename S>` syntax. This is because the first template (`T`) is already part of the class template definition.
So this would give error

```
// class and function signature
template<typename T>
class MyClass {
        template<typename S>
```

```
        some_function(T t, S s){}
}

// this gives error
template<typename T, typename S>
MyClass::some_function(T t, S s) {}

// correct way
template<typename T>
template<typename S>
MyClass::some_function(T t, S s) {}
```

Accessing pointer location with []

```
T* ptr = allocate_memory();
ptr[i];  // access content at position i in memory where ptr points to
```

That will access the values, what the address contains and not the address itself. To get the address with [] so we can iterate over the sequence of memory correctly we use the & operator `&ptr[i]`.

member functions have access to private members of other objects but with same class

Apparently member functions of a class have access to private members of other objects that are of the same class. I thought I needed an accessor method to get the ptr_, but apparently not.

```
class Iterator {
public:
    bool operator==(const Iterator& other) const {
        // Allowed: can access private member `ptr_` of another
`Iterator`
        return ptr_ == other.ptr_;
    }

    bool operator!=(const Iterator& other) const {
        // Allowed: private member `ptr_` is accessible here as well
        return ptr_ != other.ptr_;
    }

private:
```

```
    T* ptr_;  // Private member
};
```

As can be seen the member functions takes Iterator class as argument, and even though they are separate objects, the functions still have access to the private members.

valgrind

Simplest use case of valgrind to check for memory leaks, i.e. that all memory that are manually allocated to the heap by the program is freeing the memory correctly after the program termination.

```
valgrind --tool=memcheck --leak-check=yes ./complex.out
```

if you want the full error summary as well then add `-s` flag.

std::vector clear()

Weird behavior for primitive types. When used clear(), you can still access the old data via [], but not with at() because it has a bound checking. clear() only invokes destructors if the type has destructors, but primitive types don't have, so they remain. The size is then set to zero and that's it. It seems to also deprecate all references to the allocated memory.

Why can't allocate new memory right besides the current?

Because `::operator new and new` doesn't guarantee that the any subsequent memory blocks after the current ones will be available. So that's why you typically have to just reallocate new memory when resizing and let the OS decide where is appropriate.

[] for vectors can accept double or float, it will just do implicit conversion to size_t or some other integral type

As title says so you can do something like `a[1.5]` and it will round it down to 1 `a[1]`. Same goes for when initializing with a certain number you can do `std::vector<int> a(10.5)` and it will round it down to 10.

`ptr[i]` same as `*(ptr + i)`

As title says, note that it actually dereferences the pointer, so you get back the actual value stored at that memory location. Now if you want the address of that memory location you just do `&` like `&ptr[i]`, that's what I used in some of my methods when initializing new objects at certain memory locations `new (&ptr[i]) T(some_reference_to_copy_values_from)`.

Returning reference to local object is unsafe!

```
MyClass& createObject() {
    MyClass obj;
    return obj;
}
```

Lifetime of obj is only within the function, once exited after return it will cease to exist by calling it's destructor automatically.

Copy elision and return value optimization c++17 and onwards

When returning a value from a function the copy constructor is usually invoked if nothing else is specified.

```
template<typename T>
T foobar() {
        T t = T();
        return t;
}

T s = foobar();  // the return value t is created by invoking it's copy constructor
```

But in c++17 and onwards copy elision (includes RVO) is mandatory **in some cases** (not all), so that the compiler will optimize out the copy such that the copy constructor will not be called. RVO will directly construct the object in the callers memory. If objects returned for some reason can't have copy elision applied to it, then it will look for move semantics if it supports it.

When can compiler apply copy elision?

1. returning local objects like the case we saw above, then copy/move constructor is not allowed to be invoked. Here the object is instead constructed directly in the

memory space of the caller.
2. when catching by values in throwing cases, the temporary object is directly constructed in the catch block.
3. when returning directly initialized objects, almost the same as above code with foobar(), but it has `return T();` instead.

When can copy elision not be applied?

When returning different objects based on control flow like if/else

```
MyClass createObject(bool condition) {
    MyClass obj1, obj2;
    if (condition)
        return obj1; // No mandatory copy elision here
    else
        return obj2; // No mandatory copy elision here
}
```

Then the compiler doesn't know which object will be returned in compile time.

Returning object by reference

```
MyClass& createObject() {
    MyClass obj;
    return obj; // Copy elision doesn't apply, and this code is unsafe
(obj will be destroyed).
}
```

Not applicable because reference points to an existing object, not a new object, so elision doesn't work.

returning data member or global objects by value

```
class MyClass {
    MyClass member;
public:
    MyClass getMember() const {
        return member; // No copy elision for data members
    }
};
```

Since member is a member of the class it already exists, so copy elision cannot be applied. Same goes for global values/objects.

In general
Any situation where the object being returned is not a temporary or local object.

std::optional

Good to make code more robust in situations where a function can have two outcomes, success or fail outcome, and for the fail optional can provide a robust way of checking that `nullopt` instead of using a magic number.

```
template<typename T>
    std::optional<std::pair<size_t,size_t>>& find(const Vector<T>& a,
const T& x) {
        for (size_t i=0; i < a.size(); ++i) {
            if (a[i] == x) return std::pair<size_t, size_t>(i, x);
        }
    return std::nullopt;
}
```

Here we can see that for the fail outcome we don't have to return a pair object with magic numbers.

Const order matters in a function signature

```
const T foobar()  // means the returned value is of const
T foobar() const  // gives a guarantee that the function will not
perform any mutation on member variables (except mutable ones) in the
function body. Also means it can be called on const objects
```

NEVER return a reference type from a local variable

```
T& foobar() {
        T some_local_variable;
        return some_local_variable;
}
```

This is because the lifetime of a local variable is only within the function, once returned it ceases to exist, and that makes the reference to some_local_variable moot. Don't do this.

map and unordered map

Watch out for the [] notation, because if the key doesn't exist then it will create a new key for it with default value. This can be highly undesirable if you just want to check that something exists. Use find or at instead.

Fast way to reset istringstream

```
std::istringstream t;
t.clear();
t.str("");
```

Conclusion - what did I learn and what was the overall feeling?

Overall I feel like this was a good exercise just to refresh my knowledge on very basic dynamic memory management. To be reminded of what can go wrong allocating memory and initializing objects and properly deleting objects to avoid memory leaks. It's a nice exercise as it gives you more familiarity with std::vector and what kind of behaviors it has for it's functions and what strange edge cases there are for them. I chose not to look at the authors implementation of std::Vector, which was the right decision. Because of this I had to figure out myself how one could design a vector, having to go undergo the pains of coming up with the designs that makes sense I guess in a sense trains you in creating usable "APIs endpoints", because you need to think about how to make the class usable for other users. Although I shouldn't stress this point too much, as the function signature was given to me already by the cpp reference for std::vector. A little bit of exercise on clean code I guess was involved as well, but I feel like my code is not super clean. Generally the modifier methods were the hardest part for obvious reasons, they modify the memory in various ways, so that's where the dynamic memory management comes into play. Implementing different types of constructors was also very educational for me, I have done it in the past, but have forgotten exactly how to do it for like copy/move constructors. Surprisingly move semantics was the easiest to implement aside from the default constructor.

Also another cool feeling was slowly seeing the vector evolve from at first just being able to initialize it, to then adding and removing elements to and from it, to iterating it in a for loop and being able to access and modify it's contents.