PA: `vector` Template Abstract Data Type

Learning Outcomes

- Gain experience in development of generic software development using C++ classes and templates.
- Gain experience in data abstraction and encapsulation techniques by implementing abstract data types.
- Gain practice in debugging memory leaks and errors using Valgrind.
- Gain experience in developing test code for checking requirements of class templates.
- Gain practice in read and understanding code.

Read me first ...

Templates are a blessing and a curse. They have many useful properties, such as great flexibility and near-optimal performance, but unfortunately they're not perfect. As usual, the benefits have corresponding weakness. For templates, the main problem is that their flexibility and performance come at the cost of poor separation between the "inside" of a template [its definition] and its interface [its declaration]. Very often, these error messages come much later in the compilation process than we would prefer. This manifests itself in poor error diagnostics causing pages of spectacularly poor error messages to be printed to the console. The good news is that it is possible to reduce the generation of such error messages and thereby complete this assignment in a short time period by following certain basic recommendations:

1. First develop and test the class using specific types.
2. Once that works, replace the specific types with template parameters.
3. Next, it is time to test the class template with a template argument equivalent to a specific type from step 1. ***Note that for class templates, member functions are instantiated only if they're used.*** You can use this fact to save debugging time by *exercising only a small number of member functions at a time*. This means you want to eschew the use of my driver file until you're close to submitting the solution. Since my driver file exercises all of the class templates' member functions, even a minor error in a single member function could cause lengthy error messages to be generated by all the instantiations of member functions.
4. If you run into a lengthy error message, don't panic!!! First, look at the error itself and take out what is useful for you: e.g., missing operator, or something not assignable, or something that is `const` that should not be. Then find in the call stack, the innermost code in your driver where you call a member function of a class template. This will generally be the first part of the lengthy error message. Stare for a while at this code and its preceding lines because this is the most likely place where the error occurred [in my experience]. Then ask yourself, does the template argument meet the requirements and the concepts of the template to which it is applied. For example, does this type allow default construction? Does it allow its objects to be copied, to have their address, and so on.

5. Do not get scared to the point that you decide not to use templates for the rest of your life. In most cases, the problem is much simpler than the never-ending error messages make us believe. In my experience, most errors in templates can be found faster than run-time errors - with some training. And, to obtain that training you'll be implementing a class template that clones `std::vector`.

Task

The purpose of this assignment is to gain practice in the development of generic abstract data types by implementing a clone of standard library container `std::vector<T>`. Since class templates must work for current types and types yet to be invented, the direct implementation of a class template is not recommended. Instead, it is preferable to implement a few concrete classes that satisfy constraints imposed by the class template. In this spirit, the following steps are recommended to complete this assignment:

1. Begin with your implementation of concrete class `hlp2::vector` [from the previous assignment] that encapsulates a dynamic array of `int`s.
2. The copy assignment overloads in `hlp2::vector` will work perfectly in the nominal case. But what if something goes wrong? Since copy assignment involves memory allocation, it is possible the memory manager might throw an exception if there's not enough contiguous block of memory.

   When you call a member function that modifies an object's state, you typically want one of two things to happen. Usually, of course, the function fully succeeds and brings the object into its desired new state. As soon as any error prevents a complete success, however, you really don't want to end up with an object in an unpredictable halfway state. Leaving a function's work half-finished mostly means that the object becomes unusable. Once anything goes wrong you instead prefer the object to remain in, or revert to its initial state. This all-or-nothing quality of a function is formally known as *strong exception safety*.

   For your copy assignment to be strongly exception safe, you must ensure that whenever an assignment fails to allocate or copy all elements, the `hlp2::vector` object still points to the same `data` array as it did prior to the assignment attempt, and that the other data members `sz`, `space`, and `allocs` remain untouched as well.

   Class lectures introduced a programming pattern called the *copy-and-swap idiom* that can be used to guarantee the desired all-or-nothing behavior for our copy assignment. The idea is simple. If you've to modify the state of one or more objects and any of the steps required for this modification may fail and/or throw an exception, then you should follow this simple recipe:

   - *Copy* the object.
   - If necessary, *modify* the copy. All this time, the original object remains untouched!
   - Once all modifications are successful, *swap* the original object with the now fully modified copy.

   Recall that things can go wrong only during the *copy* phase since memory is being dynamically allocated and it is possible for the memory manager to not satisfy this request for memory. Therefore, it is possible to abandon the copy assignment operation if the copy phase fails, thereby leaving the original object remains as it was.

   To facilitate the copy-and-swap idiom, add the following member function to class `hlp2::vector`:

```cpp
// exchanges contents of object with contents of rhs
// where rhs is another hlp2::vector object of same type.
void swap(vector& rhs);
```

Update your copy assignments to be strongly exception safe by implementing them using the *copy-and-swap* idiom.

> ***Pay attention to values of data member `allocs` in the driver's output. Understand that `allocs` data member represents the number of "growths" or allocations of an object. However, the swap phase of copy-and-swap idiom will simply exchange all data members including `allocs`. Instead, what is required is for the `allocs` data member of the object invoking the copy assignment function to be incremented [to reflect the dynamic memory allocation performed in the copy phase].***

3. Add the following member function to class `hlp2::vector`:

   ```cpp
   // removes the last element in the vector, effectively reducing
   // the container size by one.
   void pop_back();
   ```

   Unfortunately, unlike standard library `void std::vector<T>::pop_back();`, we're unable to destroy the "popped" element. The intricacies of memory allocation/deallocation and destruction are beyond the scope of this course and will be dealt in HLP3. Instead, we leave the element alone and just reduce size `sz` by one element.

4. Use unit tests from the previous assignment plus some new tests to ensure member function `pop_back` is defined correctly.
5. Implement a second class, say `vector_str` to encapsulate a dynamic array of elements of type `hlp2::Str` [implemented in class lectures]. Adapt the previous unit tests to confirm that this new concrete class is implemented correctly.
6. Use class templates developed in lectures plus concrete class definitions `hlp2::vector` and `hlp2::vector_str` to guide your implementation of class template `hlp2::vector<T>` in header file vct.hpp.
7. Your implementation of `hlp2::vector<T>` must define *every* member function outside the class template definition. This practice will come in handy for answering final test questions.
8. The driver uses the *simple* C++ standard library template type `pair<T1,T2>` that was discussed in week 3. A very short introduction to this type is provided here.

`hlp2::vector<T>`: more restrictive than `std::vector<T>`

We only know how to dynamically allocate memory for the array encapsulated by `hlp2::vector<T>` by using `new T[n]`, where `n` is the number of elements we want to allocate. However, remember that not only does `new T[n]` allocate memory for `n` elements, but it also initializes the elements by running the default constructor for `T`. That is, expression `new[n]` does too much for our purposes: it both allocates and default initializes memory.

Since we're restricted to using `new T[n]`, we are imposing a requirement on the concrete type that will replace template type parameter `T`: clients can instantiate a concrete version of `hlp2::vector<T>` only if the concrete type has a default constructor. The standard `std::vector<T>` class imposes no such restriction:

```
// suppose SomeType is a concrete type that doesn't define a default
// ctor but defines other ctors and therefore the compiler cannot
// synthesize a default ctor
std::vector<SomeType>  v1; // ok
hlp2::vector<SomeType> v2; // error because in this assignment, we're
                           // using new[] expression to dynamically
                           // allocate memory
```

In addition, using `new T[n]` would also be unduly expensive. If we use `new T[n]`, it always initializes every element of a `T` array by using `T::T()`. If we use `push_back`, then we're copy assigning to an already default-initialized element to install the argument of `push_back`. Even worse, consider the allocation strategy that we proposed for `push_back`. This strategy implies that we'll double the size of `hlp2::vector<T>` each time we need to get more storage. We've no reason to want the extra elements initialized. They'll be used only by `push_back`, which will use the space only when we've a new element to construct in that space. If we use `new T[n]` to allocate the underlying array, these elements would be initialized regardless of whether we ever use them. In contrast, standard library `std::vector<T>` uses low-level memory management techniques relying on `std::malloc` that *only* allocate *raw* memory without initializing the memory. For this reason, `hlp2::vector<T>` is very inefficient in comparison to `std::vector<T>`.

Instead of using the built-in `new` and `delete` operators, we can do better by using standard library facilities designed to support flexible memory management. The core language itself doesn't have any notion of memory allocation, because the properties of memory are too variable to wire into the language itself. However, delving into the deeper details of memory management is beyond the scope of this introductory course. HLP3 will cover interesting and useful parts of efficient memory management.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Submission files

You will be submitting file vct.hpp that will not only contain the definition of class template `hlp2::vector<T>` but also its interface. This file should not include any header files except `<iostream>` and `<initializer_list>`. More specifically, vct.hpp should not include `<vector>` nor contain any string equivalent to `<vector>`.

Compiling, executing, and testing

Download driver vct-driver.cpp and vct-output.txt containing the correct outputs of the driver's unit tests. Follow the steps from the previous assignment to refactor a makefile and test your program.

Remember, to frequently check your code for memory leaks and errors with Valgrind. Also note that if your submission has even a single memory leak or error, F will be the maximum possible grade. You're warned!!!

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of $\textrm{g++}$ options.
    - F grade if your submission doesn't link to create an executable.
    - Your implementation's output must exactly match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). There are only two grades possible: $A+$ grade if your output matches correct output of auto grader; otherwise F.
    - A maximum of D grade if Valgrind detects even a single memory leak or error. A teaching assistance will check you submission for such errors.
    - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.

Pairs

Class template `std::pair<T1,T2>` treats two values as a single unit. This class is useful when returning two values from a function. It is used in several places in the C++ standard library including function `std::minmax` and with several container classes including `std::map` and `std::multimap`.

Class template `std::pair<T1,T2>` is defined in `<utility>` and is declared as a template `struct` from which we generate specific types:

```cpp
namespace std {
  template <typename T1, typename T2>
  struct pair {
    // member
    T1 first;
    T2 second;
    // other stuff ...
  };
}
```

We use `std::pair<T1,T2>` when we need exactly two elements and don't care to define a specific type:

```cpp
using std::pair<int, double> IntDblPair;
IntDblPair p(42, 3.14);
std::cout << "int: " << p.first << " | dbl: " << p.second << '\n';
std::cout << std::get<0>(p); // yields p.first
std::cout << std::get<1>(p); // yields p.second
```

Convenience function `std::make_pair` makes the use of pairs simple. For example, here is the outline of a function that returns a value and an error indicator:

```cpp
std::pair<double, error_indicator> my_func(double d) {
  errno = 0; // clear C-style global error indicator
  double x;
  // do a lot of computation involving d that computes x
  error_indicator ee = errno;
  errno = 0; // clear C-style global error indicator
  return std::make_pair(x, ee);
}
```

This useful idiom could be used like this:

```cpp
std::pair<double, error_indicator> res = my_func(123.456);
if (res.second == 0) {
  // use res.first ...
} else {
  // oops: error
}
```