PA: `vector` Abstract Data Type

Learning Outcomes

- Gain experience in data abstraction and encapsulation techniques using classes.
- Gain experience in encapsulating pointers, dynamic arrays, and free store in classes.
- Gain practice in debugging memory leaks and errors using Valgrind.
- Read and understand code.

Task

The purpose of this exercise is to understand how to build our own version of `std::vector` - the most useful container in the standard library. `std::vector` is a convenient, flexible, and an efficient [in both time and space] container of elements. A `std::vector` provides a sequence of elements of a given type; you can refer to an element by its index; extend the `std::vector` by using `push_back`; ask a `std::vector` for the number of its elements using `size`; and have access to the `std::vector` checked against attempts to access an out-of-range element. `std::string` has similar properties, as do other useful standard container types, such as `std::list` and `std::map`. Learning how to build our own version of `std::vector` from the basic language facilities available to every programmer allows us to illustrate useful concepts, programming, and memory management techniques. The techniques we encounter in such an implementation of a `std::vector` clone are generally useful and very widely used.

In many ways, the `Str` class we've implemented in this course has similar properties as a `std:vector`: `Str` is a container of `char`s while our `vector` is a container of `int`s. Using the knowledge gained in the implementation of `Str` class [source code is available on course web page] and using the following partial class definition in vector.hpp:

```
#include <cstddef>          // for size_t
#include <initializer_list> // for std::initializer_list<int>

namespace hlp2 {

class vector {
public:
  // type definitions for value_type, size_type,
  // pointer, const_pointer, reference, const_reference
public:
  vector();                             // default ctor
  explicit vector(size_type n);         // non-default, conversion ctor
  vector(std::initializer_list<int> rhs); // non-default, conversion ctor
  vector(vector const& rhs);            // copy ctor
  ~vector();

  vector& operator=(vector const&);
  vector& operator=(std::initializer_list<int> rhs);

  reference operator[](size_type index);
  const_reference operator[](size_type index) const;

  void reserve(size_type n);
  void resize(size_type n);
  void push_back(value_type val);

  bool empty() const;            // is container empty?
  size_type size() const;        // what is sz?
  size_type capacity() const;    // what is space?
  size_type allocations() const; // how many allocations or "growths"?

  // iterators ...
  pointer begin(); // pointer to first element
  pointer end();   // pointer to one past last element
  const_pointer begin() const;
  const_pointer end()   const;
  const_pointer cbegin() const;
  const_pointer cend()   const;

private:
  size_type sz;     // number of elements
  size_type space;  // number of elemements plus "free space"/"slots"
  size_type allocs; // number of "growths"
  pointer   data;   // address of first element
};

}
```

you must define the member functions in vector.cpp such that these functions successfully pass the unit tests implemented in the driver. If you know how to define the class functions, you can proceed to implementing the functions. If you want to get a better understanding, an explanation is provided here. In either case, heed this warning:

> ***Be particularly careful about managing memory to avoid memory leaks and errors. Since there are multiple objects in memory, allocation and deallocation is a complex process. If the automatic grader finds a memory leak or error in your submission, you will receive an F grade.***

Submission Details

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

Submission files

You will be submitting interface file vector.hpp. Complete the definition of class `hlp2::vector` and upload to the submission server. This file should not include standard library header `<vector>` nor contain the string `<vector>`.

Complete the necessary definitions of member functions of `hlp2::vector` in vector.cpp and upload to the submission server. This file should not include standard library header `<vector>` nor contain the string `<vector>`.

Compiling, executing, and testing

Download `vector-driver.cpp` and a correct input file `output.txt` for the unit tests in the driver. Follow the steps from the previous exercise 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, the automatic grader will assign an F 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 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.
   - F grade if Valgrind detects even a single memory leak or error.
   - 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.

How to "grow" `vector`?

Essential functions

Consider the partially defined class `hlp2::vector` in vector.hpp:

```cpp
#include <cstddef> // for size_t

namespace hlp2 {
class vector {
public:
  using size_type  = size_t;
  using value_type = int;
  using pointer    = value_type*;
  // other type definitions for const_pointer, reference, ...
public:
  // interface
private:
  size_type sz;   // number of elements
  pointer   data; // pointer to 1st element of free store memory block
};
}
```

> ***Hereafter, we'll refer to class `hlp2::vector` as plain class `vector`. We'll continue to use the fully qualified name `std::vector` when making references to the standard library container.***

Using `hlp2::Str` class implemented in lectures, you should be able to [easily] implement the following essential interface for `vector` in vector.cpp:

1. Default constructor: `vector::vector();`
2. Single-argument conversion constructor that allocates memory for `n` elements with each element initialized to `0`: `explicit vector::vector(int n);`
3. Another single-argument conversion constructor that allocates memory to store values in an `initializer_list<int>`: `vector::vector(std::initializer_list<int>);`
4. Copy constructor: `vector::vector(vector const& rhs);`
5. Copy assignment: `vector& vector::operator=(vector const& rhs);` and `vector& vector::operator=(std::initializer_list<int>);`
6. Destructor: `vector::~vector();`
7. Subscript operator: `reference vector::operator[](size_type index);`
8. Subscript operator overloaded on `const`: `const_reference vector::operator[](size_type index) const;`
9. Accessor function to return the number of `int` elements encapsulated by the container: `size_type vector::size() const;`.

At this point, we can:

- Create `vector`s of `int`s with whatever number of elements we want
- Copy our `vector`s using assignment and initialization

- Rely on `vector`s to correctly release their memory when they go out of scope
- Access `vector` elements using the conventional subscript notation on both the right-hand and left-hand sides of an assignment operator
- Iterate through `vector` elements

Growing a `vector`

That's all good and useful, but to reach the level of sophistication we expect [based on our experience with `std::vector`], we need to address three more concerns:

- How do we change the size of a `vector`?
- How do we specify the element type of a `vector` as an argument?
- How do we catch and report out-of-range `vector` element access similar to `at` member function of `vector`?

Specifying the element type as an argument is related to the topic of templates while reporting out-of-range element access is related to the topic of exceptions. These topics will be discussed in the second half of the course. This assessment is concerned with changing a `vector`'s size.

Class `std::vector` offers a variety of operations that can change a `vector`s size, such as `push_back`, `resize`, assignment `=`, `insert`, `erase`, `pop_back`, and so on. Consider the following code fragment that reads an unknown number of `int` values:

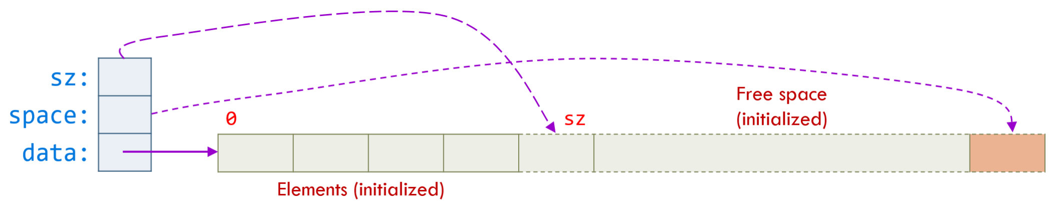
```
std::vector<int> vi;
for (int i; std::cin >> i; ) {
  vi.push_back(i);
}
```

Container `vi` begins its life with zero size. Here's one possibility for the unknown number of `int` values to be stored in `vi`. In the first iteration, `push_back` will allocate sufficient memory to store the first `int` value. In the second iteration, `push_back` finds no room for the second element and therefore has to allocate a new block of memory to store both elements. This is a simple strategy for changing size: just allocate memory for the new number of elements; copy old elements into the new memory; write the new value(s) after the old elements; and then delete the old memory.

If we resize often, that's inefficient. One way to optimize our programs is for the `vector` to anticipate such changes in size by allocating a larger-than-necessary array. The `vector` must then keep track of both the "actual" number of elements and the amount of "free space" reserved for "future expansion." The `vector` class must be augmented with the following data members:

```
namespace hlp2 {
class vector {
  size_type sz;     // number of elements
  size_type space; // number of elemements plus "free space"/"slots"
  pointer   data;  // address of first element
public:
  // interface here ...
};
} // end namespace hlp2
```

and will have the following graphical representation:

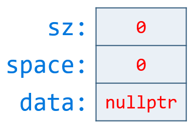


Data member `sz` represents the number of "actual" slots currently used by a `vector` while `space` represents the number of "total" available slots. Since elements are indexed from `0`, `sz` also functions as the subscript to the slot where a new `int` value must be added, while `space` functions as the subscript to one beyond the last allocated slot. The pointers shown are really `data+sz` and `data+space`.

When a `vector` is default constructed like this:

```
hlp2::vector vi;
```

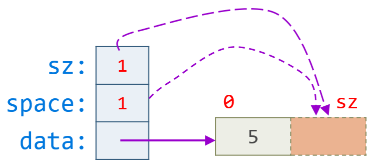
`sz` and `space` are both `0` and `data` is `nullptr`:



Now, when we use member function `push_back` like this:

```
vi.push_back(5);
```

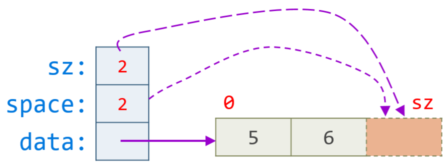
`push_back` must "grow" the empty container by one element:



Calling `push_back` again:

```
vi.push_back(6);
```

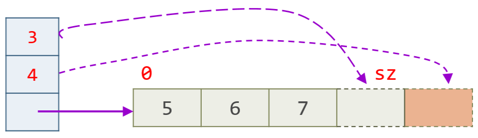
will cause `push_back` to "grow" the container since there are no slots available to insert the new `int` value `6`. This time, the function will "grow" the container by twice its previous size to two slots. This necessitates allocating a block of memory for two `int`s; copying the old element to index `0` of the new block; adding the new element with value `6` to the end of the new block; followed by deletion of the old block of memory. After the conclusion of the "growing" process, the container looks like this:



Calling `push_back` a third time:

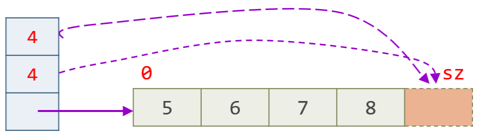
```
vi.push_back(7);
```

will require the function to "grow" the container again since there are no "empty" slots. The function will again "grow" the container by twice its previous size - from two to four slots. In the memory block, three slots are used with the fourth slot available for "future growth." This is illustrated in the following picture:



A fourth `push_back` will not require the `vector` to "grow":

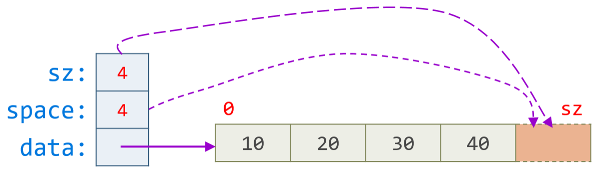
```
vi.push_back(8);
```



When a `vector` is constructed with initialization list constructor:

```
hlp2::vector vi{10,20,30,40};
```

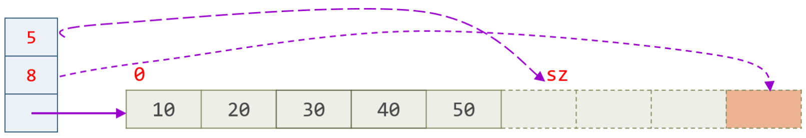
both `sz` and `space` should have the same value `4`; that is, there are no "empty" slots:



We don't start allocating extra slots until we begin inserting `int` values using member function `push_back`:

```
vi.push_back(50);
```

The function will again "grow" the container by twice its previous size - from four to eight slots. The four values from the old memory block will be copied to the first four slots in the new memory block; the fifth slot will be used to store the "new" value `50`; while three slots are available for "future growth." This is illustrated in the following picture:



`reserve` and `capacity`

The most fundamental function when we change sizes [that is, when we change the number of elements] is `vector::reserve`. This is the `vector` member function that will be used by both clients and other functions to add memory for new elements:

```
// Requests that capacity be at least enough to contain n elements.
// If n is greater than current capacity, function causes container
// to reallocate its storage increasing its capacity to n [or greater].
// In all other cases, the function call does not cause a reallocation
// and the vector capacity is not affected.
```

```cpp
void vector::reserve(vector::size_type n) {
  // 1) If n <= space, return because the function will never
  // decrease allocation
  // 2) Allocate new block of memory to store n number of ints
  // Note: Unlike with std::vector<T>, the above allocation will
  // default construct elements of reserved space. At this point, we
  // only know to dynamically allocate memory using new and new[].
  // These expressions not only allocate memory but they also initialize
  // that memory. We must learn low-level memory management functions
  // to obtain the same behavior of std::vector<T>.
  // 3) Copy old elements to new block of memory
  // 4) Delete memory containing old elements pointed to by data
  // 5) Assign to data address of first int element in new memory block
  // 6) Set space to n: space = n;
  // 7) Increment allocs by one
}
```

This assignment will measure the number of memory allocations in your submissions. To keep track of the count of memory allocations, `vector` is augmented with data member `allocs` that is initialized to zero during construction and then incremented each time function `reserve` is called:

```cpp
namespace hlp2 {
class vector {
  size_type sz;      // number of elements
  size_type space;   // number of elemements plus "free space"/"slots"
  size_type allocs; // memory allocation counter
  pointer   data;    // address of first element
public:
  // interface here ...
};
} // end namespace hlp2
```

Obviously, the amount of "free space" available for "future growth" is of interest to a client, so we [just like the standard] provide an accessor member function `size_type capacity() const;` for obtaining that information. That is, for a `vector` called `v`, `v.capacity()-v.size()` is the number of elements we could `push_back` to `v` without causing reallocation.

`resize`

Member function `void vector::resize(size_type n);` resizes the container so that it contains `n` elements. We've to handle several cases:

1. New size `n` is larger than old allocation size `space`. In this case, an automatic reallocation takes place to "grow" the container so that both `sz` and `space` are `n`. The new elements are value-initialized which for `int`s means the elements have `0` value.
2. New size `n` is larger than old size `sz`, but smaller than or equal to old allocation size `space`. In this case, the "free space" is used to accommodate the new elements that are value-initialized [in the case of `int`s to `0`]. Size `sz` changes to `n`.
3. New size `n` is equal to old size `sz`. Nothing happens.

4. New size `n` is smaller than old size `sz`. The number of elements is reduced to the first `n` elements. Unfortunately, unlike the standard library version, we're unable to destroy the elements. The intricacies of memory allocation/deallocation and destruction are beyond the scope of this course. Instead, we leave the elements alone and just reduce size `sz` to `n`.

This function requires no more than 3 lines of code with member function `vector::reserve` doing the hard work of dealing with memory [for the first case above]. The second case is easily dealt with by initializing each new element with value `0`. The third case doesn't require any work. The fourth case simply reduces the container's size to `n`.

`push_back`

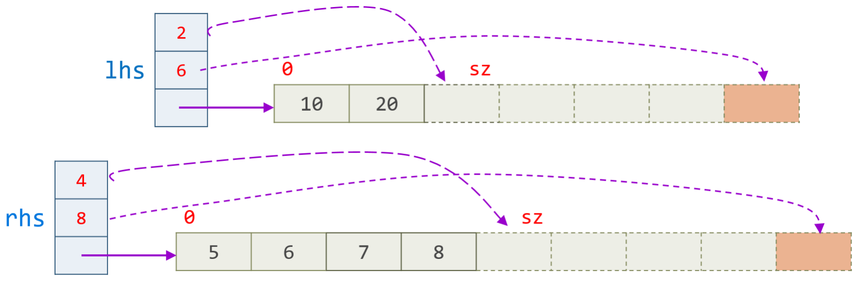
When we first think of it, member function `push_back` may appear complicated to implement, but given `reserve`, it is quite simple:

```
// Adds a new element at the end of the vector, after its current last
// element. The content of val is copied (or moved) to the new element.
// This effectively increases the container size by one, which causes an
// automatic reallocation of allocated storage space if - and only if -
// the new vector size surpasses the current vector capacity.
void vector::push_back(value_type val) {
  // if space==0 then reserve space for 1 element
  // else if sz==space then reserve space for 2*space elements
  // add val to data[sz]
  // increment sz
}
```

In other words, if we've no spare space, we double the size of the allocation. In practice, this turns out to be a very good choice for the vast majority of uses of `vector`, and that's the strategy used by most `std::vector` implementations.

Copy assignment

We know how to write copy assignment overloads. Obviously, we need to copy the elements, but what about the spare memory slots? Do we "copy" the "free space" at the end? Consider the current state of `vector` variables `lhs` and `rhs`:



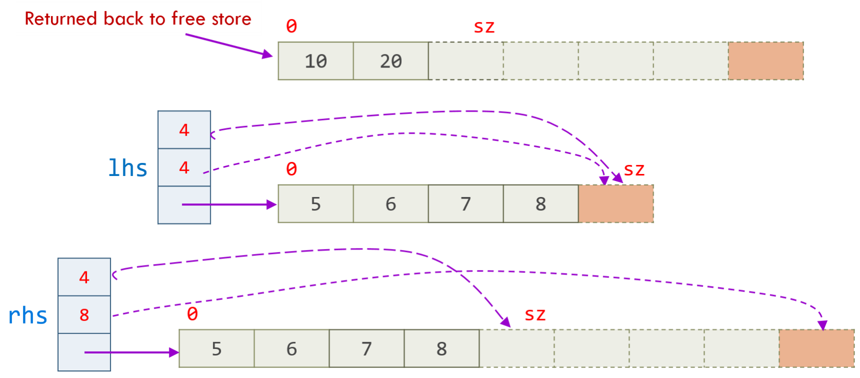
Now, consider the assignment statement

```
lhs = rhs;
```

The simplest implementation of the assignment expression would involve the following sequence of steps in `lhs`:

- `lhs` allocates exact memory needed to store the elements in `rhs` [in this example, `lhs` and `rhs` have two and four elements, respectively]. We don't copy the "unused capacity" at the end of `rhs`: `lhs` will get a copy of elements in `rhs`, but since we've no idea how `lhs` is going to be used, we don't bother with the "unused capacity" in `rhs`.
- Copy elements from old memory block to new memory block.
- Delete the old allocation.
- Set pointer `data` to the address of the first byte of the newly allocated memory block.
- Set `sz` and `space` to new values [both members will be set to `4`].

`lhs`'s state after the evaluation of the assignment expression will look like this:



This implementation is correct but looking closely we realize that the current copy assignment definition may cause a lot of redundant allocation and deallocation. What if left operand `lhs` has more elements than right operand `rhs`? What if `lhs` has the same number of elements as `rhs`? In both cases, memory allocation and deallocation is not necessary; instead, the elements encapsulated by `rhs` can be copied into the memory already available to `lhs`. This optimization is left as an exercise.

> ***Caution: The unit tests use the simplest implementation described above and do not take into account any of the optimizations described here.***