Queens College, CUNY, Department of Computer Science
Object-Oriented Programming in C++
CSCI 211/611
Summer 2018
Instructor: Dr. Sateesh Mane

due date Friday, July 20, 2018, 11.59 pm

Homework: Classes: construct, copy, assign, destroy

- **Experience with other classes has demonstrated that in many cases the source of difficulty is not the mathematics or the programming.**
- **The source of difficulty is the English (understanding the text).**
- **If you do not understand the words in the lectures or homework, THEN ASK.**
- **If you do not understand the concepts in the lectures or homework, THEN ASK.**
- **Send me an email, explain what you do not understand.**
- **Do not just keep quiet and then produce nonsense in exams.**

- **Consult your lab instructor for assistance.**
- **You may also contact me directly, but I cannot promise a prompt response.**
- Please submit your inquiry via email, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.
- Please submit one zip archive with all your files in it.
  1. The zip archive should have either of the names (CS211 or CS611):

     ```
     StudentId_first_last_CS211_hw_classes2.zip
     StudentId_first_last_CS611_hw_classes2.zip
     ```

  2. The archive should contain one "text file" named "hw_classes2.[txt/docx/pdf]" and one cpp file named "Vec_int.cpp" etc.
  3. Note that a text file is not always required for every homework assignment.
  4. Note that not all questions may require a cpp file.

General information

- You should include the following header files, to run the programs below.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <cmath>
```

- If you require additional header files to do your work, feel free to include them.
- **Include the list of all header files you use, in your solution for each question.**
- The questions below do not require complicated mathematical calculations.
- If for any reason you require help with mathematical calculations, **ask the lab instructor or the lecturer.**

Class `Vec_int`

- **We shall write a class `Vec_int` which simulates some functions of `vector<int>`.**

```
class Vec_int {
public:
  // to do

private:
  int _capacity;
  int _size;
  int * _vec;
};
```

- The class has three private data members.
- **We shall follow a common industry practice and attach an underscore "_" for the names of class data members.**
- The data member `_vec` is a pointer to `int` and is a dynamically allocated array.
- The data members `_capacity` and `_size` have the same meanings as for `vector<int>`.
  1. The value of `_capacity` is the length of the dynamically allocated array `_vec`.
  2. The value of `_size` is the number of elements in `_vec` which are populated with data..
  3. Obviously `_capacity >= _size >= 0` and it is our responsibility to make sure our code always satisfies this relationship.
- Obviously initially `_capacity=0` and `_size=0` and `_vec=NULL`.
- We shall write public and private methods (including constructors and a destructor).

Q1 `const` methods: accessors and `front()` and `back()`

- The simplest to write are the accessor methods.
- **Write two accessor methods `capacity()` and `size()`.**
  1. They are both public.
  2. They are both `const`.
  3. They both have return type `int`.
  4. It should be obvious what data value each function returns.
- **Write two methods `front()` and `back()`.**
  1. They are both public.
  2. They are both `const`.
  3. They both have return type `int`.
- **front()**
  1. This method returns the first of the populated elements in the array.
  2. If `_size <= 0` then return 0.
  3. If `_size > 0` then return `_vec[0]`.
  4. Obviously if `_size > 0` then `_vec` is not NULL.
- **back()**
  1. This method returns the last of the populated elements in the array.
  2. If `_size <= 0` then return 0.
  3. If `_size > 0` then return `_vec[_size-1]`.
  4. Obviously if `_size > 0` then `_vec` is not NULL.

Q2 Non-const methods: `clear()` and `pop_back()`

- **Write the methods `clear()` and `pop_back()`.**
  1. They are both public.
  2. They both have return type `void`.
  3. They are not `const` because they change the value of the class data member `_size`.
- **clear()**
  1. All it does is set `_size=0`.
  2. *That's it.*
  3. We do not perform any memory deallocation. The destructor will do that.
- **pop_back()**
  1. If `_size > 0` then decrement its value by one: `_size = _size - 1`.
  2. You can also decrement by writing `_size--`.
  3. *That's it.*
  4. The return type is `void` so we do not return the value of the popped array element.
  5. Nor do we perform any memory deallocation.

Q3 Private methods: `allocate()` and `release()`

- Now we come to the details of dynamic memory management.
- **Write two methods `allocate()` and `release()`.**
  1. They are both **private.**
  2. They both have return type `void`.
- **allocate()**
  1. This method dynamically allocates an array for `_vec`.
  2. The length of the array is `_capacity`.
     (a) *What if the value of* `_capacity` *is zero?*
     (b) Then we will attempt to allocate an array of zero length.
     (c) **Software designers have to think of details like this.**
  3. **Do the following:**
     (a) **If `_capacity <= 0` then set `_vec = NULL`.**
     (b) **If `_capacity > 0` then use operator `new`.**

```
_vec = new int[_capacity];
```

- **release()**
  1. This method deallocates the dynamically allocated memory.
  2. If `_vec` is NULL, do nothing and exit.
  3. **If `_vec` is not NULL then release the memory.**

```
delete [] _vec;
_vec = NULL;
```

  4. **As a safety precaution, set `_vec = NULL` at the end.**
  5. **This is to avoid problems in case `release()` is calld twice by accident.**

Q4 Constructors

- Now we can write the class constructors.
- We shall write three constructors, one default and two non-default.

```
Vec_int();
Vec_int(int n);
Vec_int(int n, int a);
```

- Just for practice, write all the constructors **non-inline.**
- **Default constructor.**
  1. Use the member initialization list and set everything to zero or NULL.

     ```
     Vec_int::Vec_int() : _capacity(0), _size(0), _vec(0) {}
     ```

  2. The function body is empty because there is nothing to do.
  3. ***If you are not comfortable with the member initialization list, you do not have to use it.***
  4. Initialize `_capacity = 0`, `_size = 0` and `_vec = NULL`.
- **Non-default constructor** `Vec_int::Vec_int(int n).`
  1. We have to guard against bad input $n \leq 0$.
  2. **If** $n \leq 0$ **then initialize** `_capacity = 0, _size = 0` **and** `_vec = NULL.`
  3. Else if $n >$ do the following.
     (a) **Initialize** `_capacity = n` **and** `_size = n.`
     (b) **Call** `allocate().` This will allocate the memory and initialize the array to zero.
- **Non-default constructor** `Vec_int::Vec_int(int n, int a).`
  1. We have to guard against bad input $n \leq 0$.
  2. **If** $n \leq 0$ **then initialize** `_capacity = 0, _size = 0` **and** `_vec = NULL.`
  3. Else if $n >$ do the following.
     (a) **Initialize** `_capacity = n` **and** `_size = n.`
     (b) **Call** `allocate().` This will allocate the memory and initialize the array to zero.
     (c) However, we must set the values of the array elements to the input value a.
     (d) **Write a loop and set** `_vec[i] = a` **for** $0 \leq i <$ `_capacity.`

Q5 Big Three: copy, assign, destroy

- The Big Three are the copy constructor, assignment operator and the destructor.

```
Vec_int(const Vec_int &orig);
Vec_int& operator= (const Vec_int &rhs);
~Vec_int();
```

- Write them all **non-inline.**
- **Destructor**
- The destructor is easy. Just call `release()`. There is nothing else to do.

```
Vec_int::~Vec_int()
{
  release();
}
```

- **Copy constructor**
  1. The non-inline function signature is as follows.

     ```
     Vec_int::Vec_int(const Vec_int &orig) { // etc
     ```

  2. **Note that we only need to copy elements up to the size in the original object, not its capacity (which may be very large).**
  3. **Hence initialize as follows;** `_capacity = orig._size`**.**
  4. **Initialize** `_size = orig._size`**.**
  5. **Call** `allocate()`**.** This will allocate memory with the correct length.
  6. Next, perform the deep copy of the values of the array elements **up to the size.**
  7. **Write a loop and copy** `_vec[i] = orig._vec[i]` **for** $0 \leq i <$ `_size`**.**

- **See next page(s).**

- **Assignment operator**

  1. The non-inline function signature is as follows.

     ```
     Vec_int& Vec_int::operator= (const Vec_int &rhs) { // etc
     ```

  2. **First perform the test for self-assignment.**
  3. **Note that we only need to copy elements up to the size in the original object, not its capacity (which may be very large).**
  4. **Hence assign as follows;** `_capacity = rhs._size`**.**
  5. **Assign** `_size = rhs._size`**.**
  6. **Call** `release()`**.** This deallocates the memory in the destination object.
  7. (The copy constructor does not have to perform this step. Notice that the assignment operator shares code with the copy constructor and the destructor.)
  8. **Call** `allocate()`**.** This allocates fresh memory with the correct length.
  9. Next, perform the deep copy of the values of the array elements **up to the size.**
  10. **Write a loop and copy** `_vec[i] = rhs._vec[i]` **for** $0 \leq i <$ `_size`**.**
  11. **Remember to** `return *this`**.** Exit the function.

Q6 Populate the array: `push_back(...)`

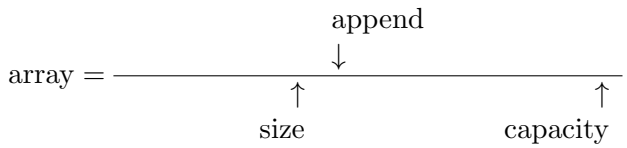
- We now arrive at the task of populating the array.
- **Declare the method `push_back(int a)`.**
- The method is somewhat complicated and should be written **non-inline.**

```
void push_back(int a);                          // declaration in class
void Vec_int::push_back(int a) { // etc         // non-inline function
```

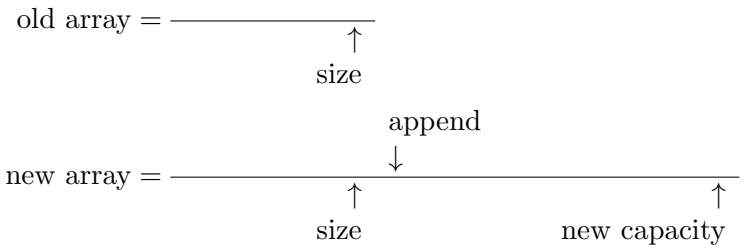
1. The method is public.
2. The return type is `void`.
3. The method is not `const` because it changes the values of the class data members.

- This is the overall task to achieve.

1. We wish to "append" a data item to the set of populated elements in the array `_vec`.
2. Hence the value of `_size` will increment by one (the number of populated elements).
3. However, we must check if the capacity (length of the array) is large enough to do this.
4. It the capacity is large enough, there is no problem. Just set the value of the relevant element in the array and increment the value of `_size` by one.



5. It the capacity is *not* large enough, the algorithm is more complicated.
   (a) We must allocate fresh memory for a new array.
   (b) We must copy the values of the populated data from the old array to the new array.
   (c) We must deallocate the old array, to avoid a memory leak.
   (d) Then we set the input data value using an element in the new array and increment the value of `_size` by one.



- There are two cases to consider (i) `_size < _capacity`, (ii) `_size == _capacity`.
- We begin with the easy case `_size < _capacity`.
  1. **Set the value `_vec[_size] = a`.**
  2. Increment the value of `_size` by one: `_size = _size + 1`.
  3. You can also increment by writing `_size++`.
  4. Exit the function.
- Next we treat the more diffiult case `_size == _capacity`.
  1. Increase the value of the capacity.
     (a) **If `_capacity == 0` then set `_capacity = 1`.**
     (b) **Else if `_capacity > 0` then double its value.**
     (c) You can write `_capacity = _capacity * 2` or `_capacity *= 2`. Both are correct.
  2. **Declare a temporary pointer to save the address of the current array.**

     ```
     int *oldvec = _vec;
     ```
  3. **Call `allocate()`.**
     (a) This allocates fresh memory with the increased value of `_capacity`.
     (b) The pointer `_vec` points to the new memory address.
     (c) We have saved the address of the old array in the pointer `oldvec`, so it is not lost.
  4. If `oldvec` is not NULL, do the following.
     (a) Copy the values of the array elements from `oldvec` to `_vec`.
     (b) **Write a loop and set `_vec[i] = oldvec[i]` for $0 \leq i <$ `_size`.**

     ```
     for (int i = 0; i < _size; ++i) {
       _vec[i] = oldvec[i];
     }
     ```
     (c) *We copy only the populated data.* Hence the upper limit is `_size`, not `_capacity`.
     (d) We no longer require the old array, so deallocate it using operator `delete []`.

     ```
     delete [] oldvec;
     ```
  5. Either way, if `oldvec` was NULL or not, perform the same actions as for the other case.
  6. Set the value of `_vec[_size]` and increment the value of `_size` by one.

     ```
     _vec[_size] = a;
     // increment value of _size by one
     ```
- *We are done.* Exit the function.

Q7 Read/write data: `at(...)`

- The method `push_back` appends a data value to the end of the ppopulated data.
- We need a method to read/write the existing data for any populated element in the array.
- **Declare the method `at(int n)`.**
- The method should also be written **non-inline.**

```
int& at(int n);                              // declaration in class
int& Vec_int::at(int n) { // etc             // non-inline function
```

  1. The method is public.
  2. **The return type is a reference `int&`.**
  3. This is because we wish to both read **and write** the value of the data element.
  4. This will be explained below.
  5. Hence the method is **both an accessor and a mutator.** Such things can exist.
  6. Because the method is a mutator, it is not `const`.
  7. It is possible to write separate read-only (accessor) and write-only (mutator) methods, but we shall not do so.

- Clearly, the method accesses the array element indexed by the value `n`, i.e. `_vec[n]`.
- **We must perform validation tests to check if the value of n is valid.**
- First we deal with the case where the value of n is valid.

  1. **If $n \geq 0$ and also $n <$ `_size`, return `_vec[n]`.**
  2. The returned value is a **reference to `_vec[n]`.**
  3. **Hence the calling application can change the value of `_vec[n]`.**
  4. That is why the return type is a reference `int&` and the method is a mutator.

- Next we deal with the case where the value of n is *not* valid.

  1. This is the case if $n < 0$ or $n \geq$ `_size`.
  2. **Declare a null pointer. Return a dereference to the NULL pointer.**

```
int *pnull = NULL;
return *pnull;
```

  3. As astonishing as it looks, the above code is legal.
  4. There will be a memory fault if the calling application tries to read/write the reference.
  5. *Why do this?* Simply because the return type is `int&`, hence we must return *something*.
  6. The return type is not a Boolean `bool` so we cannot return `false`.
  7. The return type is not a pointer `int*` so we cannot return a NULL pointer.
  8. We also cannot "return 0" because we must return a *reference*, not a number.
  9. The alternative is to throw an exception, which has its own problems.

Q8 Class declaration

- The declaration of the class `Vec_int` looks like this.
- You should have correctly working function bodies for all the class methods.

```
class Vec_int {
public:
  Vec_int();
  Vec_int(int n);
  Vec_int(int n, int a);

  Vec_int(const Vec_int &orig);
  Vec_int& operator= (const Vec_int &rhs);
  ~Vec_int();

  int capacity() const;
  int size() const;

  int front() const;
  int back() const;

  void clear();
  void pop_back();
  void push_back(int a);

  int& at(int n);

private:
  void allocate();
  void release();

  int _capacity;
  int _size;
  int * _vec;
};
```

Q9 Functions `reverse(...)` and `print(...)`

- Let us write some functions which use the class `Vec_int`.
- These are external functions, not methods of the class.
- **Write a function to reverse the order of the array elements in a `Vec_int` object.**

```
void reverse(Vec_int &v);
```

  1. The return type is `void`.
  2. The input argument is a reference to `Vec_int`.

- Note that we only reverse the elements up to the size of the array, not its capacity.
- Remember that only the array elements $i = 0, \ldots,$ `size` $- 1$ are populated.
- **Because `reverse` is an external function, not a class method, it cannot access the private data directly.**
- **We must employ the public accessor methods.**
- The function body is as follows.

  1. Declare a local variable `int n = v.size()`, i.e. use the public accessor method.
  2. If $n \leq 1$ then return. There is not enough data to reverse anything.
  3. Run a loop and swap the array elements `v.at(i)` and `v.at(n-1-i)`, for $i = 0, \ldots, n/2$.
  4. (If n is odd there will be one element in the middle which is not swapped.)
  5. We must use the public method `at` because we cannot access the array `_vec` directly.
  6. Exit the loop, return and exit the function.

- **Write a function to print the array elements in a `Vec_int` object.**

```
void print(Vec_int v);
```

  1. The return type is `void`.
  2. The input argument is a `Vec_int` object, simply to test call by value.

- The details of how you wish to display the data are up to you.
- *Don't spend too much time on fancy formatting.*

Q10 Main program

- **Write a main program to test your code, including `reverse` and `print`.**

- **Your class should be able to perform all the tasks in the homework for vectors (for the data type `int`), except the function `resize()`.**